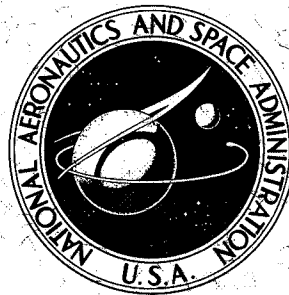


**N A S A T E C H N I C A L
R E P O R T**



NASA TR R-338

NASA TR R-338

**PROGRAM ANALYSIS - A PROBLEM
IN MAN-COMPUTER COMMUNICATION**

by Joseph Green

*Electronics Research Center
Cambridge, Mass. 02139*

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION • WASHINGTON, D. C. • JUNE 1970

| | | | |
|--|--|---|-----------------------|
| 1. Report No. NASA TR R-338 | 2. Government Accession No. | 3. Recipient's Catalog No. | |
| 4. Title and Subtitle Program Analysis--A Problem in Man-Computer Communication | | 5. Report Date June 1970 | |
| | | 6. Performing Organization Code | |
| 7. Author(s) Joseph Green | | 8. Performing Organization Report No. C-89 | |
| 9. Performing Organization Name and Address Electronics Research Center Cambridge, Mass. | | 10. Work Unit No. 125-23-02-36-25 | |
| | | 11. Contract or Grant No. | |
| 12. Sponsoring Agency Name and Address (National Aeronautics and Space Administration) | | 13. Type of Report and Period Covered Technical Report | |
| | | 14. Sponsoring Agency Code | |
| 15. Supplementary Notes | | | |
| 16. Abstract <p>This report presents a display oriented scheme to help the higher level language computer programmer to debug and analyze programs. Timed visual interpretive execution and a variety of user instituted functions permit an informative dialogue between the man and the computer. A prototype implementation of the system using FORTRAN as the higher level language is described, and the results are discussed from the point of view of the system developer as well as the point of view of the user. Also treated is the problem of what a time-sharing system should do to make graphical input tablets useful devices in a time-shared environment.</p> | | | |
| 17. Key Words Debugging •Graphics •High level languages •Interpretive compiler •Display •Interactive | | 18. Distribution Statement Unlimited | |
| 19. Security Classif. (of this report) Unclassified | 20. Security Classif. (of this page) Unclassified | 21. No. of Pages 129 | 22. Price * \$3.00 |

*For sale by the Clearinghouse for Federal Scientific and Technical Information
Springfield, Virginia 22151

TABLE OF CONTENTS

| | Page |
|---|------|
| TABLE OF CONTENTS..... | iii |
| LIST OF FIGURES..... | v |
| SUMMARY..... | 1 |
| CHAPTER I - INTRODUCTION - THE PROBLEM OF PROGRAM ANALYSIS | |
| 1.1 Trends in Computer Usage | 2 |
| 1.2 The Present Solution to Program Analysis..... | 3 |
| 1.3 Interactive Graphics and Dynamic Analysis..... | 4 |
| CHAPTER II - GRAPE THEORY | |
| 2.1 Basics..... | 5 |
| 2.1.1 Unifying Concepts..... | 5 |
| 2.1.2 Hardware..... | 8 |
| 2.1.3 Display Set-up..... | 11 |
| 2.2 Source Program Execution..... | 11 |
| 2.2.1 Introduction and Example..... | 11 |
| 2.2.2 Non-executable Statements..... | 14 |
| 2.2.3 Expressions..... | 14 |
| 2.2.4 Arithmetic Statements..... | 17 |
| 2.2.5 Control Statements..... | 19 |
| 2.2.6 Logical Statements..... | 19 |
| 2.2.7 Iterative Loops..... | 21 |
| 2.2.8 Subroutine Calls..... | 21 |
| 2.2.9 Input/Output..... | 23 |
| 2.2.10 Speed Control and Related Statements..... | 24 |
| 2.3 Execution Functions..... | 25 |
| 2.3.1 Changes with Execution Stopped: Null, →, ROLL, RESTART, SAVE..... | 25 |
| 2.3.2 Changes During Execution: Breaklines, Display and Set Variables, □ . | 30 |
| 2.4 Editing Functions..... | 37 |
| 2.4.1 Changing Code: >, m, ^, Overwriting, & | 37 |
| 2.4.2 Checking Code: Line Numbers, Set Variables, Cross References, Compiler Diagnostics, UNUSED, TRASH..... | 39 |
| 2.5 Special Situations..... | 47 |
| 2.6 User's Manual..... | 55 |
| CHAPTER III - GRAPE PRACTICE | |
| 3.1 Novel Hardware and Software Building Blocks..... | 56 |
| 3.1.1 Required..... | 56 |
| 3.1.2 Provided..... | 57 |
| 3.2 Implemented GRAPE -- User Side..... | 60 |
| 3.2.1 Basics..... | 60 |
| 3.2.2 Source Program Execution..... | 60 |
| 3.2.3 Execution Functions..... | 61 |
| 3.2.4 Editing Functions..... | 66 |
| 3.2.5 User's Manual..... | 68 |

| | Page |
|--|--|
| 3.3 | Implemented GRAPE -- System Side..... 69 |
| 3.3.1 | Concepts in the Programming..... 69 |
| 3.3.2 | Structure of the System..... 69 |
| 3.3.3 | Data Structure..... 70 |
| 3.3.4 | Other System Information..... 71 |
| 3.3.5 | Graphical Input Tablet Interface..... 74 |
| 3.3.6 | Language..... 75 |
| 3.4 | Results of Implementation..... 77 |
| 3.4.1 | Teaching Tool/Analytic Tool..... 77 |
| 3.4.2 | Debugging Tool/Programming Errors..... 78 |
| CHAPTER IV - TIME-SHARING HIGH SPEED GRAPHICAL INPUT DEVICES | |
| 4.1 | Central Problems..... 83 |
| 4.2 | What Must Be Provided..... 84 |
| 4.3 | What Ought To Be Provided..... 85 |
| 4.3.1 | Feedback..... 85 |
| 4.3.2 | Data Compression..... 86 |
| 4.4 | What Might Be Provided..... 87 |
| 4.4.1 | General..... 87 |
| 4.4.2 | Data Reduction..... 88 |
| 4.4.3 | Improved Buffering..... 92 |
| 4.4.4 | Mode Analysis..... 93 |
| CHAPTER V - EXPANSION AND CONTRACTION OF GRAPHICAL PROGRAM ANALYSIS | |
| 5.1 | More Powerful Hardware and Software..... 95 |
| 5.2 | Less Powerful Hardware..... 98 |
| 5.2.1 | Display, Light Pen, and Keyboard Without Graphical Tablet..... 98 |
| 5.2.2 | Keyboard and Display Without Light Pen.....102 |
| 5.2.3 | Weaker Refresh Display.....102 |
| 5.2.4 | Remote (Low Speed) Storage Display.....105 |
| 5.3 | Conclusions.....109 |
| REFERENCES111 | |
| APPENDIX A - GRAPE PROGRAM LISTING.....113 | |

LIST OF FIGURES

| Figure | | Page |
|--------|---|------|
| 1 | Hardware Setup..... | 9 |
| 2 | Tablet..... | 10 |
| 3 | Speed Control..... | 10 |
| 4 | Display Setup..... | 12 |
| 5 | Typical Display..... | 13 |
| 6 | Display Messages When the Speed Is At STOP..... | 28 |
| 7 | Condensing A Program Loop..... | 33 |
| 8 | Three Displayed Variables..... | 41 |
| 9 | Four Displayed Variables..... | 41 |
| 10 | Error Message..... | 43 |
| 11 | Statement Cross Reference..... | 44 |
| 12 | Variable Cross Reference..... | 45 |
| 13 | Unused Code..... | 46 |
| 14 | Trash File..... | 48 |
| 15 | Display Output..... | 50 |
| 16 | ARRAY (24)..... | 52 |
| 17 | ARRAY (4,3)..... | 53 |
| 18 | ARRAY (50,9)..... | 54 |
| 19 | Display Setup..... | 62 |
| 20 | Language Syntax..... | 63 |
| 21 | Sample Statements..... | 64 |
| 22 | Typical Display..... | 65 |
| 23 | GRAPE Block Diagram..... | 73 |
| 24 | Program With An Infinite Loop..... | 80 |
| 25 | Execution Of An Infinite Loop..... | 81 |
| 26 | Display Messages When The Speed Is At STOP..... | 101 |
| 27 | Speed Control..... | 103 |
| 28 | Display Messages When The Speed Is At STOP..... | 104 |
| 29 | Display Setup..... | 107 |

PROGRAM ANALYSIS — A PROBLEM
IN MAN-COMPUTER COMMUNICATION

By Joseph Green*
Electronics Research Center

SUMMARY

The writer of higher level language computer programs has few tools available to help him debug a program that he has written or to help him understand a program that someone else has written. This report presents a display oriented scheme to aid in higher level language program analysis.

Using a computer driven refresh display for output and a graphical tablet for input, the GRAPE (Graphical Analysis of Program Execution) system presents information to the user about the source program as it is executing. About forty source program statements are displayed on the screen at one time, and as each statement executes, that statement is briefly modified to show the results of the execution. Besides the source program execution, execution functions such as the removal of blocks of statements from the screen and editing functions such as the setting of program variables comprise the system.

A prototype version of GRAPE has been implemented using a subset of FORTRAN as the higher level language. The results of the implementation are described from the point of view of the system developer, as well as the point of view of the user. Included is a description of how the system must be modified to adapt to changes in the hardware configuration, ranging from a three-dimensional display down to a remote (low speed) storage display with keyboard input.

One other topic treated in this report is the problem of what a time-sharing system should do to make graphical input tablets useful devices in a time-shared environment.

*Submitted to Harvard University as Ph. D. thesis in Applied Mathematics May 1969.

I. INTRODUCTION - THE PROBLEM OF PROGRAM ANALYSIS

1.1 Trends in Computer Usage

Although computer programming is commonly taught in schools and businesses and is a skill required by many people, relatively little attention has been paid to the problems of programming. More study and analysis has been given to the problem of tape operating systems, for example, than has been given to the more general problem of how to make a tape operating system, or any other program, run successfully once the basic design is worked out.

The initial programming of a procedure used to control a computer has been aided by the development of computer languages designed for specific uses, but the manipulation and analysis of a computer program in order either to correct an improperly written program or to understand a basically correct program remains exceedingly difficult. The latter problem especially needs attention for purposes of teaching programming and because it is frequently true that it is easier to rewrite a program than to modify a program written by someone else. As yet, few tools exist to help an individual understand the structure of a given computer program.

Several mutually reinforcing trends in computer utilization are accentuating this lack. First is the trend toward having non-professional computer users. If highly repetitive, time consuming jobs are to be automated, it may make sense to have an expert program for maximum efficiency. And if an engineer or physicist decides to learn to use computers, he already has much of the knowledge necessary to be adept at computer programming. But as uses for computers are developed in fields which are not computer oriented, the users become less proficient. Programming becomes more of a chore, and some automatic assistance is more desirable.

Second, and coupled with the above, is the trend toward using the computer as an occasional tool rather than as the focal point of a project. Programs are written quickly, used once or twice, and then discarded. As a result, the time spent preparing programs is extremely high in proportion to the time spent interacting with the computer.

Third is the increasing use of higher level languages such as FORTRAN, MAD, ALGOL, PL/I. Programming is easier to learn and programs are faster to write when higher level languages are used. However, the writer of an assembly language

program has an understanding of the execution, or lack of execution, of his program which the writer of a higher level language program lacks.

A fourth trend is the rapid increase in computer installations. With more computers in existence, the chances are greater that someone else has already written a program which will do almost what one wants his program to do.

As people who are not professional programmers begin to use higher level languages to quickly write or modify computer programs to assist them in their work, it becomes more and more desirable to have effective ways to deal with and understand computer programs. This is true both for the relatively experienced programmer with a complicated problem and for the novice programmer with a simple problem, if either of them must sit at a desk and simulate the operation of the computer because his program gave no output or did not work.

1.2 The Present Solution to Program Analysis

The higher level language programmer, whether he is learning to program, trying to correct a complicated program he has written, or trying to modify a program written by someone else, would like to know how the program operates. Instead, what he has been getting for communication with his program is vestiges of the operation of his program from which he attempts to reconstruct what must have happened. When he is most unlucky, he gets nothing to guide him except the fact that his program failed. When he is luckier, he gets a memory dump. Memory dumps at the time of program failure constitute one extreme of description. None of the program operation is described, just the final results of its operation. At the other extreme is a machine cycle by machine cycle indication of the state of the computer. This sounds frightening and indeed it is, but countless programs written in a higher level language have been debugged at the console of a small computer because memory dumps and one day turnaround made batch process debugging a less desirable alternative.

The so called trace features of some compilers (ref. 1) are a decided improvement on this mode of operation. Yet they do not make debugging a simple task. If the user gets the message that WEIGHT has just been set to one million, he may indeed immediately realize that his error is in the only statement that sets the value of X which is then used to compute WEIGHT. But a more frequent situation is that a variable used as a counter is being set incorrectly. To discover this error requires that the results of all arithmetic statements be listed, but more important,

it requires that the individual mentally put the message "I = 4" into the framework of his entire program. A good analysis system must as much as possible supply that framework.

Using these same features on a time-sharing system means that one can be selective about what is traced and can hopefully locate errors with several short runs rather than be inundated with trace data. But this also does not supply the context of the program in which a particular statement is executed. In addition, it is useless for finding errors in which a program decision is based on a comparison of variables X and Y instead of X and Z. And it is useless for giving insight into the way an unfamiliar program is structured.

A major fault in all the methods above, except for console analysis, is they all leave the user trying to figure out what happened instead of letting him see what is happening. A major fault of trace features, because of time and space limitations, is that they are highly selective in the information they supply. A major fault of console analysis and memory dumps is that they convey information in a language which may be unfamiliar or unknown to the user (assembly language if he is lucky, octal numbers if he is not) and, in any case, is not the language in which he is attempting to communicate with the computer.

1.3 Interactive Graphics and Dynamic Analysis

The higher level language programmer trying to analyze a program would be aided in his understanding if he could see the effect of all the statements in the program. The data transmission rate of a teletype is much too slow to make this feasible, but a computer driven display which is being refreshed through the computer output channel can show new information far faster than the human observer can digest it (ref. 2). While a teletype can produce perhaps 15 characters per second, displays can produce up to 75,000. This means that there is no need for the output to be selective or for words to be abbreviated. There is ample time to display all the information which the programmer would like to see, including any background information which might help his understanding. Because display images disappear after they have been on the screen and because there are display techniques to emphasize certain portions of the picture, one can avoid giving the programmer so much material that he cannot locate the most relevant data.

The use of high speed displays to aid higher level language programmers is the intent of GRAPE. GRAPE is a programming

system for the GRaphical Analysis of Program Execution. It permits the programmer to see his higher level language program execute, and it permits him to interact with that program as it is executing. In turn, it operates under whatever executive system is used on the computer. The GRAPE system is described in Section II of this paper, and a particular implementation of GRAPE is described in Section III.

If interaction between user and programmer is to occur, there must be an input device. For certain aspects of the GRAPE system characters must be input to the computer, and the device best suited to this is the graphical input tablet (ref. 3). A powerful computer is needed to handle graphical devices such as displays and tablets, but interactive use of this equipment means that the user may spend only a small fraction of his time exercising the computer and the bulk of his time thinking. The desire to bring users "on line" with equipment which they will use for only a few seconds each minute often leads to time sharing. Therefore, the executive system under which GRAPE is running may be a time sharing system. Section IV presents a general approach to the problems of time sharing high speed graphical devices such as two dimensional tablets and three dimensional Lincoln Wands (ref. 4).

Section V is primarily concerned with the changes that must be made in GRAPE when it is used with hardware configurations that range from three dimensional displays to remote storage displays.

II. GRAPE THEORY

2.1 Basics

2.1.1 Unifying Concepts.— Section I described the inability of present language translator and computer executive systems to provide a superior capability for an individual, either computer expert or novice, to correct or modify a higher level language* computer program. GRAPE provides that capability. It is a system for displaying, on a computer driven display, information about a program while that program is being executed by the computer. This information is used to assist in debugging the program if it has errors, to assist in understanding the program if modifications are being attempted, and to assist in understanding programming if one is learning to program. The program being analyzed is written in a higher level language, and the nature of the displayed information is such as to make much easier the normally time consuming and tedious task of understanding what a given program actually does.

*FORTRAN, ALGOL, MAD, PL/I are typical higher level languages.

The basic idea of GRAPE is that the computer executes and displays source program HLL (higher level language) statements at a rate suited to the observer and shows on the display the result of that execution in the language of the source program. The observer can at any time halt execution, make changes to the source program, and restart execution. This, in itself, is useful for certain functions such as teaching programming. Yet, there are several other concepts which are also vital in GRAPE that make it an effective programming tool for both novice and expert.

First, all communication between man and computer in GRAPE is in the language of the program, a higher level language. In addition, the unit of information from the computer is the same unit which the man chose to use, a single HLL statement.

Second, continuous feedback is given to the man. Extensive feedback can only be given because of the high data rates which the computer driven display can produce and the human eye/brain can accept and interpret. The precise form of this feedback is chosen to maximize information content while minimizing the mental load on the man.

Third is the idea of context. A particular piece of information will be meaningful to the man only if he has the framework for the information being presented. The high data rates of the display permit it to present the context surrounding a particular HLL program statement. Further, the new piece of information will be emphasized over its surrounding context.

Thus the display must show at one time at least as much of the source program being analyzed as a man would normally look at during a desk simulation of the program operation. One is supposed to feel that the entire program is essentially in front of him, and if the window through which one is looking is too small this feeling is lost. There is a critical size for this window below which a major purpose of GRAPE is not served, and therefore one may have to tolerate some display flicker in order to achieve this size. This critical size is influenced by several factors. A single line of code in a language such as BCPL (ref. 5) is usually more complex than a single line of FORTRAN code, and fewer lines of code may be needed to reach the critical size. The increased sharing of computer programs among users is leading to the shortening of program units and the writing of subprograms to perform a program function even if that function is invoked only once. Such subprograms are easier to debug, to change, to share, and to understand. Display oriented program analysis may both benefit from and contribute to that trend. Investigation of numerous HLL programs indicates that for long programs twenty-five lines of displayed code is a minimum and forty is more desirable. Of course the text must be in

large enough characters to be easily readable. In the description of GRAPE, techniques for increasing the size of the useful window will be shown.

Fourth, since the speed of execution of the HLL program must be suited to the observer, there must be some speed control which provides for a wide range of execution speeds so that one can easily adjust or halt program execution. This control should have only one degree of freedom, should move continuously, should be stable when set to a value, and should be trivial to operate. It should not be something which falls down if one lets go of it, even if the speed is not thereby affected.

Fifth, beside the speed control flexibility, there must be other execution capabilities. One must be able to manipulate the execution of the HLL program so that all the information one might want from this program is readily available.

Sixth, one must have an editing facility which is sufficiently good that minor editing is not a major chore. It is character recognition software which recognizes graphical tablet input as being specific letters that permits such editing.

Seventh, the system must be easy to learn and natural to use. Visual output as provided by a computer driven refresh display and pencil-and-paper input as provided by a graphical input tablet offer far more capability in this direction than other computer peripherals.

Because a display can present so much data in a fraction of a second, there is no need to have the user learn abbreviations for everything. Both novice and expert can let the system supply him with complete information which will disappear as soon as he no longer needs it.

The number of separate GRAPE functions has been kept to a minimum. Several of these functions are taken from previously learned human activities, and all functions have immediate feedback and non-destructive effects. Therefore GRAPE is self-instructing to the extent that one learns about it by using it. The user's manual for GRAPE is one page long. There is almost nothing that need be memorized by the user in order to use the system.

Using GRAPE, the programmer is dealing directly with his source language program. His hands and eyes are on his program. It is his program that is executing, not GRAPE, and not the binary result of a compilation. He is not talking about his program to an intermediary operating system, and he is not trying to glean information about events that have already happened.

2.1.2 Hardware.- The computer peripheral hardware for GRAPE consists of a refresh display and a graphical input tablet. A picture of the hardware setup is given in Figure 1.* The face of the display should be as large as possible, since the characters must be large enough to be easily readable and there should be as many lines of text as possible on the screen. The display should be powerful enough to show several hundred characters without serious flicker. The graphical input tablet need not be more powerful than the weakest of the currently available ones. A twelve inch square writing surface and a five millisecond interrupt rate is sufficient. No special purpose computer hardware is necessary for the operation of GRAPE.

The graphical input tablet is the sole input medium, and the tablet stylus is used to control the source program execution speed, to point to messages on the display, and to edit source program code. To control the speed the stylus is supported vertically on a low mount which runs about eight inches along the closer left side of the face of the tablet, as in Figure 2. Sliding the stylus forward and backward changes the speed. No information about the speed is displayed on the screen. The stylus rests loosely in the ring in Figure 3. Although the stylus can be lifted from the slide at any time, the other input actions can only be performed after the stylus has been moved to the STOP end of the scale. Moving the speed control to STOP halts all GRAPE action and so the speed control serves as a ready "panic button." In fact, the entire design of the speed control is meant to provide security to the user.

A graphical tablet stylus has a switch in its penpoint, and a slight pressure on the stylus will serve to trigger this switch. One prints characters on the tablet by pressing on the stylus and drawing the character. Stylus pressure is also used as part of the speed control scheme.

To permit writing on the tablet, the speed lever mount must be on the left side of the tablet for right-handed people and on the right side for left-handed people.

As will be seen in Section V, GRAPE can also be used with other hardware configurations. The source program execution and the execution functions are not seriously affected if a keyboard and a light pen are substituted for the graphical input tablet,

*A rear projected transparent tablet would be more appropriate although tablet users seem to have little trouble adjusting to the situation of writing on the tablet and seeing the "ink" on the display.

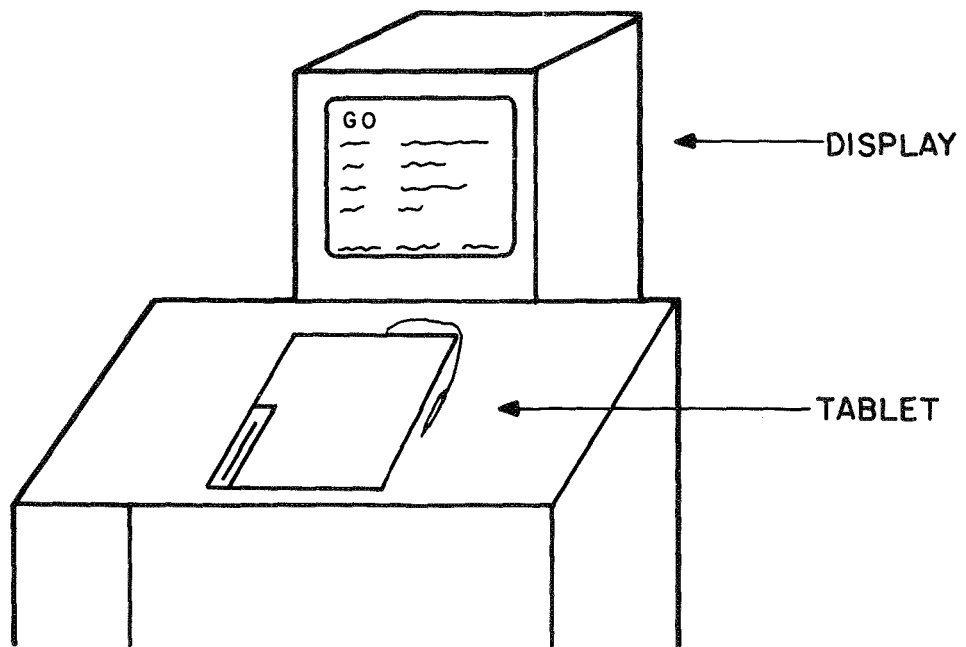


Figure 1.- Hardware Setup

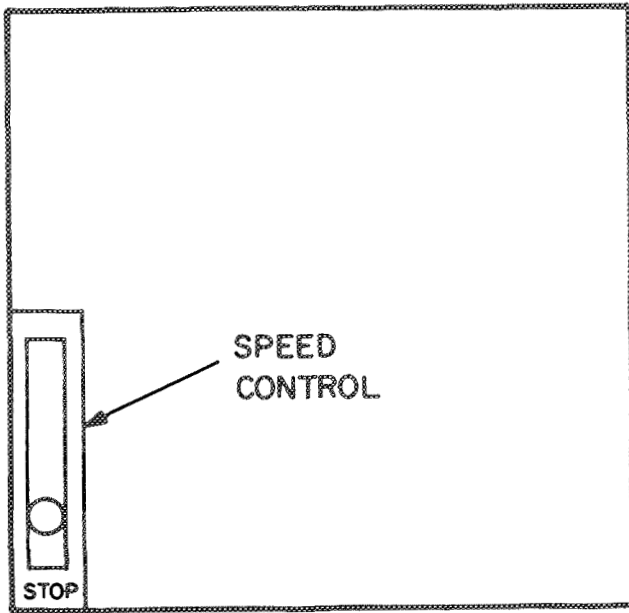


Figure 2.- Tablet

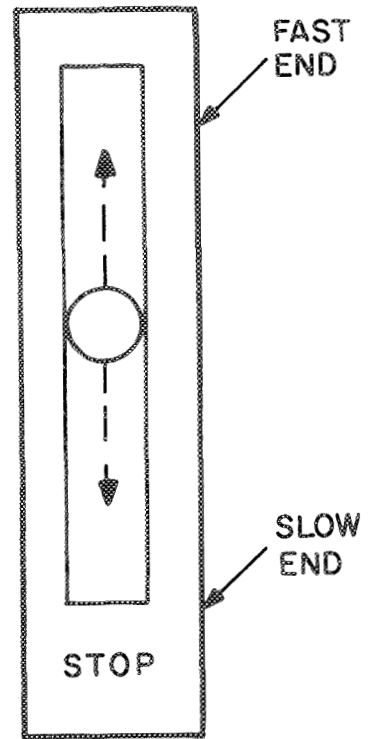


Figure 3.- Speed Control

although editing becomes less natural. The light pen can be removed with a loss primarily in the naturalness of the speed control. If a low speed storage display is substituted for the refresh display, some reworking of the display techniques is necessary, but GRAPE can still be used for effective program analysis.

2.1.3 Display Setup.- The setup of the information which is on the display during operation of GRAPE is in Figure 4. I have taken FORTRAN (ref. 6) as a typical HLL throughout the description of GRAPE because it is undoubtedly the most commonly used HLL. The bulk of the screen is taken up with about forty lines of the programmer's HLL program. Just to the left of the code and to the right of the statement labels, one character width of space is left for entering and displaying certain special symbols which affect how program execution will take place. On the left of the screen beside each displayed line of code is a line number. At the bottom of the screen is space for the displaying of HLL program variables. At the top of the screen messages are displayed appropriate to whatever is presently occurring. For example, when the HLL program is being executed the only message displayed is "GO". A typical display is shown in Figure 5.

2.2 Source Program Execution

2.2.1 Introduction and Example.- We now look at the information that is displayed as the user HLL source program is executing. During execution, the only message displayed at the top of the screen is GO. As long as that is the only message, the user is sure that his program is operating. At the bottom of the screen, information requested by the user may be displayed, as will be described in Section 2.3. The source program code fills the rest of the screen. The entire screen except for the single source program statement which is presently being executed is displayed at a uniform brightness. That single statement, including its line number, is displayed more brightly. What in fact is displayed brightly is code which is designed to indicate the result of each source statement, but which has as few symbols different from the original code as possible, and which requires as little human eye movement as possible. This technique places the smallest possible burden of interpretation on the user. Therefore, for example, brightening is used rather than cursors. Whenever it can be done, brightening is the sole change in the source code. For example, if during source program execution the statement

```
GO TO 140
```

is reached, then for the length of time which GRAPE is using to execute each source statement that line of code will appear as

```
GO TO 140
```

| | | | |
|-------------------------|----------------------------|--------------------------|------------------|
| GO | | | |
| 41 | <u>Messages</u> | _____ | |
| 42 | <input type="checkbox"/> | _____ | |
| 53 | | _____ | |
| 54 | | _____ | |
| 55 | | _____ | |
| 56 | | _____ | |
| 57 | | _____ | |
| <u>Line Numbers</u> | <u>Special Symbols</u> | <u>Program Codes</u> | |
| 58 | <input type="checkbox"/> | _____ | |
| 64 | | _____ | |
| 65 | | _____ | |
| ~~~~~ | = | ~~~~~ | <u>Variables</u> |

```

GO

30          SUBROUTINE DELETE
31 C        □ SPECIFICATIONS
36          DUMMY = LINE1
37 130      DUMMY = POINT(DUMMY)
38          IF (VIZ(DUMMY).EQ.3) GO TO 130
39          IF (VIZ(DUMMY).GE.4) CALL ERROR
40          VIZ(LINE1) = 3
41          CALL REGEN (0)
42          RETURN
43          END

LINE1 = 184

```

Figure 5.- Typical Display

2.2.2 Non-executable Statements.- GRAPE spends the same amount of time at a non-executable statement that it does at any other statement. One reason for doing this is that if non-executable statements were performed faster, the flow on the screen would not be as smooth and would be harder to follow. A more important reason is that the most frequent non-executable statements which are imbedded in executable code and reached many times during program execution are the comment statement and the CONTINUE. A user is as likely to be interested in looking at imbedded comment statements or a CONTINUE statement as he is at any other statement.

Non-executable statements are displayed by brightening (the line number and) the code which indicates their functions. Typical Fortran statements as they would be displayed during execution are

```

C          COMPUTE THE ANGULAR ACCELERATION

DIMENSION  ROWSUM(8),  COLSUM(15),  HEAT(8,15)

INTEGER    HEIGHT, WEIGHT
and other specification statements

40  FORMAT  (I3, 2F6.1)

120  CONTINUE

SUBROUTINE  SETUP    (INIT, MAXMUM, MINMUM)

FUNCTION    CUBIC    (A,B,C,D)

```

ALGOL (ref. 7) statements such as ARRAY or BEGIN are treated the same way.

2.2.3 Expressions.- Before showing how other statements are displayed, we must look at how the evaluation of expressions is displayed. The evaluation of expressions takes more than one execution cycle. When execution of an expression begins, all code on the screen physically below the statement containing the expression drops down one line, and the temporary results of the execution are displayed on that line just below the expression. The general rule for evaluation of expressions is that one level of operations is executed and displayed in each execution cycle. As a first example we will use the arithmetic expression to the right of the equal sign in the arithmetic statement

$$I = J + K$$

Execution of the above expression takes two execution cycles and happens as follows (if J is 7 and K is 4). The code on the display screen

$$W = 2$$

$$I = J + K$$

$$M = I + W$$

becomes

$$W = 2$$

$$I = J + K$$

$$= 7 + 4$$

$$M = I + W$$

and then becomes

$$W = 2$$

$$I = J + K$$

$$= 11$$

$$M = I + W$$

The following example, with subscripted variables, takes four steps. Let $K = 6$, $\text{INT}(13) = 8$, $\text{INT}(17) = -2$.

The code

$$I = \text{INT}(K+7) + \text{INT}(K+11)$$

becomes

$$I = \text{INT}(K+7) + \text{INT}(K+11)$$

$$= \text{INT}(6+7) + \text{INT}(6+11)$$

which becomes

$$I = \text{INT}(K+7) + \text{INT}(K+11)$$

$$= \text{INT}(13) + \text{INT}(17)$$

then

$$I = \text{INT}(K+7) + \text{INT}(K+11)$$

$$= 8 + (-2)$$

and then

$$\begin{aligned} I &= \text{INT}(K+7) + \text{INT}(K+11) \\ &= 6 \end{aligned}$$

In more complicated situations involving imbedded expressions, functions, and subscripts, the rule still holds that evaluation proceeds one level at a time. In the following example, only the steps in the evaluation are shown. $J = 4$; SQRT is a library function; X is a subscripted variable; $K = 7$; $X_6 = 25$; $X_5 = 3$.

The code $I = J * (\text{SQRT}(X(K-1)) + X(K-2))$
starts as $= 4 * (\text{SQRT}(X(7-1)) + X(7-2))$
becomes $= 4 * (\text{SQRT}(X(6)) + X(5))$
then $= 4 * (\text{SQRT}(25) + 3)$
then $= 4 * (5 + 3)$
then $= 4 * (8)$
and finally $= 32$

The execution and display of a logical expression follows the same pattern as an arithmetic expression. Logical operators are treated as shown in the following example. LV1 and LV2 are logical variables.

The code

IF (LV1 .AND. LV2) GO TO 30

may become

IF (LV1 .AND. LV2) GO TO 30
(t .AND. f)

which becomes

IF (LV1 .AND. LV2) GO TO 30
(f)

Relational operators are treated similarly.

The code

```
IF (I .EQ. J) GO TO 30
```

may become

```
IF (I .EQ. J) GO TO 30
(7 .EQ. 3)
```

which becomes

```
IF (I .EQ. J) GO TO 30
( f )
```

One final example will demonstrate that a logical expression containing both kinds of operators and containing arithmetic expressions is no more complicated, only longer. Again just the changes from step to step are shown. $I = 4$, $J = 9$, $K = 12$.

The code `IF (I+J .GT. K .AND. I+J .LT. K+5) GO TO 30`

starts as `(4+9 .GT. 12 .AND. 4+9 .LT. 12+5)`

becomes `(13 .GT. 12 .AND. 13 .LT. 17)`

then `(t .AND. t)`

finally `(t)`

This expression took four steps to evaluate. If one were beset by an expression which took ten or twenty steps to evaluate, which was inside a program loop, and which was guaranteed to be correct, one could eliminate the tedious displayed evaluation in any of several ways. Using the condense feature described in Section 2.3, one could set it to indicate only the results of the evaluation. Also using the condense feature one could remove the statement from the screen entirely. Finally, using the speed control, one could pass through the steps of the evaluation as fast as desirable.

2.2.4 Arithmetic Statements.— Three things must be added to the previous description of arithmetic expressions in order to evaluate arithmetic statements. While the expression part of an arithmetic statement (the part to the right of the equal sign) is being evaluated to an arithmetic value, the part to the left is being evaluated to an address. One example of the full evaluation of an arithmetic statement will suffice to demonstrate this. $I = 6$, $J = 7$, $A_{9,7} = 10$.

The code

$$A(I+3,J) = A(I+3,J) + J + 1$$

starts as

$$A(I+3,J) = A(I+3,J) + J + 1$$
$$A(6+3,7) = A(6+3,7) + 7 + 1$$

becomes

$$A(I+3,J) = A(I+3,J) + J + 1$$
$$A(9,7) = A(9,7) + 7 + 1$$

then

$$A(I+3,J) = A(I+3,J) + J + 1$$
$$A(9,7) = 10 + 7 + 1$$

finally

$$A(I+3,J) = A(I+3,J) + J + 1$$
$$A(9,7) = 18$$

If the variable to the left of the equal sign is an externally defined variable, that is, if it is in COMMON in FORTRAN or global in ALGOL, it is displayed brightly during the statement execution.

On the matter of precision of arithmetic values during displayed execution, GRAPE only gives two significant digits of accuracy to variables whose values are being displayed. In general, greater precision than one percent is not required during the moving evaluations and six or seven characters for each displayed value would make the evaluation hard to follow. There are other ways in GRAPE to display variables to complete accuracy as will be described in Section 2.3. If the value of a variable is between -99 and +99, it is displayed as a two digit integer. If the value is outside this range it is displayed as two significant digits, an upward arrow to indicate exponentiation, and the appropriate power of 10. For example, 5321 is displayed as $53\uparrow 02$.

Alignment of successive steps during evaluation of both arithmetic and logical statements must be considered. Whenever possible, in the displayed code parentheses and operators (arithmetic, relational, logical) appear directly under their

counterparts in the original code. Values replacing variables appear directly under the variables. Values replacing sub-expressions appear directly under the first operator in the sub-expression being replaced, as in the expressions evaluated earlier. When necessary, as when a variable such as X is replaced by a value such as 53+02, some or all of the expression will have to be moved horizontally to the right. As soon as possible in successive evaluation steps, the alignment will revert to the original.

2.2.5 Control Statements.- The display of these statements indicates the results of transfers or conditional transfers. Examples will be the clearest way to demonstrate this. The first example below is clear. In the second example let J equal 3. In the third example let M be statement 22. In the fourth example e is an arithmetic expression which has been evaluated by the methods above to be exactly zero. The intensification of the appropriate statement number is the final step in the displayed execution of the arithmetic IF statement. An ALGOL switch statement would also have the evaluation of an expression preceding the indication of the transfer.

Examples: GØ TØ **37**

GØ TØ (18, 46, 11, 46), J

GØ TØ M, (43, 9, **22**)

IF (e) 19, **14**, 19

More than any other group of statements the control statements show, even without the use of a display, how the operation of the source program is clearly defined within the source code itself.

The ASSIGN statement is included here because of its association with control statements. It also is simply to display.

ASSIGN **30** TØ N

2.2.6 Logical Statements.- In a logical IF statement, IF (e) S, the expression is evaluated as described in Section 2.2.3 above. If the result is true the next execution cycle is the first cycle appropriate for statement S. If the result is false the next cycle is the first cycle for the statement immediately following the logical IF.

Take the code

110 C TEST FØR CØMPLETIØN

111 IF (INDEX.EQ.J) GØ TØ 40

112 GØ TØ 50

If INDEX = 6, J = 7 this entire block, including line numbers, executes as

```
110 C      TEST FØR CØMPLETIØN
111        IF (INDEX.EQ.J) GØ TØ 40
112        GØ TØ 50
```

Then

```
110 C      TEST FØR CØMPLETIØN
111        IF (INDEX.EQ.J) GØ TØ 40
           (6      .EQ.7)
112        GØ TØ 50
```

Then

```
110 C      TEST FØR CØMPLETIØN
111        IF (INDEX.EQ.J) GØ TØ 40
           (      f      )
112        GØ TØ 50
```

Finally

```
110 C      TEST FØR CØMPLETIØN
111        IF (INDEX.EQ.J) GØ TØ 40
112        GØ TØ 50
```

If INDEX = 7, J = 7 the block executes as

```
110 C      TEST FØR CØMPLETIØN
111        IF (INDEX.EQ.J) GØ TØ 40
112        GØ TØ 50
```

Then

```
110 C      TEST FØR CØMPLETIØN
111        IF (INDEX.EQ.J) GØ TØ 40
           (7      .EQ.7)
112        GØ TØ 50
```

Then

```
110 C      TEST FØR CØMPLETIØN
111        IF (INDEX.EQ.J) GØ TØ 40
           (      t      )
112        GØ TØ 50
```

Finally

```
110 C      TEST FØR CØMPLETIØN
111        IF (INDEX.EQ.J) GØ TØ 40
112        GØ TØ 50
```

2.2.7 Iterative Loops.- Automatic loops also have simple display versions. For purposes of display all three steps inherent in loops (initialize, increment/modify, test) take place at the statement which defines the loop, even though compilers often put the latter two at the end of the loop. This is the natural way to think of loop execution.

As long as source program execution is within a DO loop, or either type of ALGOL FOR loop, the value of the index variable remains displayed beside the loop statement. Thus

```
DØ 30 I = 1,40,1 14
```

This statement is executed each time through the loop by incrementing the index variable, I. The final execution of this statement sets I to 41, and the next statement executed is the one after statement 30. Checking to see if a transfer statement has taken execution outside the range of a program loop, so that the value of the loop can be removed from the screen, is something which compilers rarely do, but which is valuable within the slow compilation and execution scheme used by GRAPE.

2.2.8 Subroutine Calls.- The execution of a call to a subroutine takes two display cycles. The actual execution of the subroutine takes place between the two cycles. During the first cycle the code physically below the subroutine call drops down one line, as in evaluation of an expression, and on the display appears the present values of the variables being used as inputs to the subroutine. Thus, if the first, second and third arguments in the following example were inputs to subroutine NEWTON, the third being a logical variable, the code

```
CALL NEWTON (OLDVAL, 5.3, LOGVAR, NEWVAL, ITERNO)
```

would appear during the first execution cycle as

```
CALL NEWTON (OLDVAL, 5.3, LOGVAR, NEWVAL, ITERNO)
```

```
TO NEWTON (21      , 5      , t      , NEWVAL, ITERNO)
```

During the second execution cycle, after the execution of the entire subroutine at full computer speed, the code on the display drops down one more line and the results of the subroutine execution are displayed on the next line.

```
CALL NEWTON (OLDVAL, 5.3, LOGVAR, NEWVAL, ITERNO)
```

```
TO NEWTON (21      , 5      , t      , NEWVAL, ITERNO)
```

```
FROM NEWTON (OLDVAL, 5.3, LOGVAR, 81      , 30      )
```

The display of both TO and FROM lines simultaneously permit the user to see quickly if any variables have been used for both input to and output from the subroutine or if any variables have been used for neither. These situations often, though not always, indicate incorrect subroutine usage.

Actually, matters are not quite so simple with subroutine calls. First, only single valued variables will have their values displayed. For multi-valued variables such as entire FORTRAN arrays the name of the array is intensified in the TO and/or FROM lines. The same is done if the argument is a function name. As with the precision of displayed variables, the capability to display entire arrays is discussed in later sections. Second, if one of the arguments is an expression, the expression is evaluated in the manner of 2.2.3 above before the subroutine is called. Third, a compiler cannot tell which arguments in a call are inputs to the subroutine.* The best way to handle this is to permit the user to specify to GRAPE which arguments in each subroutine are inputs. This can be done in the statement immediately following the SUBROUTINE statement, and it can be done in a comment statement so that the normal HLL compiler will still accept the source program. However since this, or any other such scheme, involves additions to the HLL code or to some "system cards," there is a default condition. With no information to the contrary, GRAPE displays all arguments as input arguments.

*This is true for FORTRAN but not true for some other higher level languages.

Fourth, arguments returned from subroutines pose a similar problem. If the subroutine were being executed interpretively GRAPE could keep track during execution of the subroutine of which arguments were modified. If, however, the subroutine is to run at full machine speed, and if, as is often true, the argument transfer vector is used as a set of indirect addresses, GRAPE has no way of knowing which arguments are returned. The solutions offered for the input case can be used here, and in the default case all arguments are displayed as outputs. All and only the output arguments should have changed in value between the two cycles.

The above implies that the code for subroutines does not normally appear on the screen. How to bring the subroutine onto the screen is discussed in 2.3. Display of the SUBROUTINE statement was given in 2.2.2 above. The RETURN statement is displayed by intensifying the whole word

RETURN

2.2.9 Input/Output.— Most input and output presents no special problems to GRAPE. Execution of an input or output statement takes one execution cycle, unless subscript evaluation or other expression evaluation occurs first. During the cycle devoted to the input/output itself, code below the I/O statement drops down one line and the values of any single valued variables which are being read or written are displayed. The device type is also displayed.

The code

```
WRITE (N,20) I1, I2, (MAT(K), K = 1,10)
```

might become

```
WRITE (N,20) I1, I2, (MAT(K), K = 1,10)
```

```
WRITE (1,20) I3, 6, (MAT(K), K = 1,10)
```

At the same time, the input or output actually occurs. The display of output arrays can be requested as in 2.3.

If the I/O device is fast enough, the data transfer may take place in less than one GRAPE execution cycle. In that case there is no delay in program execution. However, if the input or output takes longer, e.g., if the input is from a keyboard or a teletype, GRAPE waits for completion of the I/O. The values of variables on the intensified line of an input statement cannot, of course, be filled in until the input has taken place. No values are displayed for binary input/output, as opposed to character I/O.

The one form of input/output which presents problems is display output, since the display screen is already filled with GRAPE information. (Graphical tablet input does not cause the same difficulties since GRAPE does not use the tablet during the execution of a source program input statement, except as described in 2.2.10.) The frequency of occurrence of display output is increased by the fact that when one is analyzing a program one might well want to do all output on the display, rather than on a relatively slow teletype or on a non-readable magnetic tape. This situation is discussed and solutions presented in 2.5.

2.2.10 Speed Control and Related Statements.- The speed control hardware has already been described. By sliding the lever back and forth the user adjusts the speed, in execution cycles per second, at which his source program operates. Most types of statements take one execution cycle to execute. Others, such as subroutine calls, take two cycles to execute. The existence of either arithmetic or logical expressions within statements increases the number of execution cycles used to display the statement in execution. Experimentation has shown that the slowest speed which is useful for continuous execution is one cycle every four seconds. Speeds slower than this can be induced by single cycle execution. When the lever is at the near end of the slide (Figure 3) execution stops. Pressing down on the lever while it is in that position, thus tripping the stylus switch, causes the execution of one cycle. This action can be repeated as often as desired. Moving the lever away from the STOP position restarts continuous execution.

The fastest speed which now appears useful if one is watching the execution is forty times as fast, ten cycles per second, and occurs when the lever is at the end of the slide away from STOP. That speed is too fast for the user to see individual statements execute, but one can see the area of the source program that is being executed, and one can see changes in information being displayed at the bottom of the screen. Speeds faster than this can be induced by pressing down on the lever while it is at the fast end of the scale. This completely eliminates delay between execution cycles, although the results of each statement are still displayed on the screen. Nothing intelligible can be seen at this speed; one would use it only if he knew there was a statement in the program such as teletype input or a breakpoint which would cause execution to stop. Even with no GRAPE delay, as long as each source statement is being executed for display, execution will be two orders of magnitude slower than if the statement had been compiled by a "real" compiler. For this and other reasons there is a way to make sections of the source program run even faster. It is described in 2.3.

Intermediate positions of the speed lever cause intermediate execution speeds. None of the speeds except STOP are marked on the speed control.

As implied above there are source program statements other than input which cause execution to cease.* When execution reaches one of these statements, continuous source program execution stops regardless of the position of the speed lever, and the message at the top of the screen is changed

from GO to STOP

The statement which caused the halt remains intensified. The user's next actions will depend on the nature of the statement.

If the statement is a temporary halt, the user can continue with execution by pulling the speed control to the STOP position and then pushing it away. Alternatively he could move it to STOP and then perform the functions described in the rest of Section 2. The FORTRAN PAUSE statement is intensified as

PAUSE

If the statement is a permanent halt, all the user can do is move the speed control to STOP and perform some of the GRAPE functions described below. The two FORTRAN statements which are permanent halts are also intensified by having the entire statement be brightened. They are

STOP AND END

2.3 Execution Functions

2.3.1 Changes with Execution Stopped. - No editing or program modification can be done while the HLL program is in the process of executing. However, when the speed control is pulled to STOP, changes to the program can be made. In 2.3 changes are described that affect the manner in which the HLL program is executed or the manner in which it is displayed. Changes to the HLL code itself will be described in 2.4. Some of the functions introduced are familiar ones, such as breakpoints in the program, but even there the method of invoking the function will be new.

Some of these functions take their effect while execution is stopped. When execution is started again they leave no trace. The first of these is the null function: simply pushing the speed control away from STOP. When this is done execution of the HLL program continues from exactly where it had stopped. Since the slow end of the speed control is nearest STOP, no great

*These statements were used frequently on user operated computers. With the advent of batch processing and time-sharing they became less useful. GRAPE returns the good parts of user operated computers, including these statements.

changes suddenly happen when the control is moved. Therefore, one can easily execute just a statement or two, then stop and think. By pressing down the stylus, and causing single cycle execution, one has even finer control over execution. This is a good time to note that the execution of a statement does not actually occur until the statement has finished its display cycles. This is clearly true for transfer statements, but it is also true that as long as an arithmetic statement is still being displayed, even if the speed is at STOP, the value of the left side variable has not yet been changed. The exceptions to this are the subroutine call, which is executed between the TO and FROM cycles, and input/output statements, which are executed while they are displayed.

It may be that one wishes to start execution at some statement in the HLL program other than where execution stopped. One reason might be that the section of code about to be executed cannot be executed properly; perhaps it has not been completely written, or perhaps it requires some input which is not yet ready. Another reason is that as a result of watching the execution, one has stopped execution, has made some changes to the code, and now wishes to rerun the few statements which were changed. The second execution function permits this. To start execution at a particular statement on the display face, one draws an arrow to that statement in the column reserved for execution functions (Figure 4). Execution immediately begins at that new statement, but since the speed is at STOP, only the first execution cycle will take place. If, while execution is stopped, one draws the arrow indicated below:

```

118      IF (I.NE.3) GO TO 20
119      →J = 16 + I
120      K = 0
121      GO TO 45
122  20  CONTINUE

```

then execution will begin at line 119 and the display will read

```

118      IF (I.NE.3) GO TO 20
119      J = 16 + I
          J = 16 + 3
120      K = 0

```


121 GO TO 45

122 20 CONTINUE

Moving the speed control away from STOP will continue with the execution at line 119. Since all statements take at least one cycle to execute, no harm is done by pointing at the wrong line. (Remember that the graphical input stylus which is being used for writing is also the speed control lever).*

There is no need for GRAPE to ever display the arrow. The instant that GRAPE knows an arrow was drawn, the new statement will be properly intensified. While the arrow is being drawn, the computer operating system must provide the necessary feedback to the user to help him draw his characters properly (see Section IV).

Whenever the speed is STOP, not only does the displayed message, GO, change to STOP, but other messages appear at the top of the screen as in Figure 6. These messages are all function buttons; touching any of them with the tablet stylus will cause an appropriate activity to occur. The full set of messages for this situation can also be seen in the User's Manual (2.6) next to "speed set to STOP."

The third execution function which affects the HLL program only while execution is stopped is ROLL. Only forty or so lines of program code are visible on the screen at any one time. For longer programs there must be a way to get specific sections of the source program on the screen so that one can edit them or can perform execution functions such as "start here" on them. The message ↑ROLL↓ permits this. Pressing the down arrow with the stylus rolls the program past the display, displaying code closer to the end of the program. As long as the arrow is pressed, the program keeps rolling, except that once the end of the program is on the screen the down arrow has no effect.** Pressing the up arrow has the opposite effect; the code rolls towards the beginning of the program.*** During source program execution this

*The choice of written symbols for this and the other execution functions is somewhat arbitrary, since these functions are new, whereas the symbols for the editing functions are the standard typographical ones. An arrow for "start here" seems appropriate.

**If ROLL is difficult to understand, one can think of the credits rolling across the screen after a television show.

***Which way an arrow moves a picture depends on whether one thinks of the screen as moving over the picture, or of the picture as moving under the screen. In the present situation, the former seems more satisfying.



Figure 6.- Display Messages When The Speed Is At STOP

is handled automatically. When control transfers to a program statement which is not presently on the screen, the window moves to the new part of the program making that statement the third line on the screen.

The rate of the roll depends on where along the stem the arrow is touched. The end at the arrow tip is the fast end; the program will roll at about forty lines per second. This high speed is necessary so that one can get to the other end of a long program. Touching the end away from the tip rolls the program at about one line per second. Intermediate spots along the stem cause intermediate rates. As with the other execution functions, no harm is done if the function is used incorrectly, and so no special instruction in the use of ROLL is necessary. The other arrow can be used to return the earlier picture. Note, incidentally, that if the speed control is pushed away from STOP after a long roll has taken place, execution still begins where it had halted, and the correct page of display code is brought onto the display.

One other message, the fourth execution function, affects what section of code is displayed on the screen. Touching RESTART puts the first forty lines of the program on the screen. It also resets all program variables to their initial values, usually "undefined". This is the only function in GRAPE which has an immediate irreversible effect. To prevent accidental disaster, when RESTART is pressed the screen goes blank except for the question:

DO YOU REALLY WANT TO RESTART

YES ☐

NO ☐

Touching the NO box returns the previous picture.

A button with a different function is the SAVE button. This button terminates GRAPE operations, saves the present version of the HLL program in the computer file system, and returns control to the computer operating system. SAVE calls the operating system in such a way that the present status of the machine is not destroyed and GRAPE can be restarted in exactly the situation in which it was terminated. If GRAPE plus appropriate input/output operating routines are the sole programs in the computer and there is no operating system, control remains in GRAPE after the source program is saved.

The final two function buttons displayed when the speed control is set to STOP are used in conjunction with editing. Details of their use will be described in the section on editing.

2.3.2 Changes During Execution.- Now several functions which affect the program display as execution is taking place will be examined. The first of these is the break line. Drawing a hook over a line number, thus

```

140 IF (I+3.GT.J*7) TBAR=TBAR+1

```

will cause execution to stop, whenever that statement is about to be executed. The break line and the line number are brightened. The GO at the top of the screen changes to STOP. In principle, this is similar to inserting a temporary halt statement, such as FORTRAN PAUSE, before the given statement, but the break lines are easier to see and eliminate than are the PAUSE's. To continue execution after a stop because of a break line, one pulls the speed lever to STOP and then pushes it away. Any number of break lines can be inserted in a program. Two consecutive execution cycles in a program are shown below.

```

118 GO TO 40
119 I = I + 1
120 40 M = I * 2 + SQRT (J)

```

and

```

118 GO TO 40
119 I = I + 1
120 40 M = I * 2 + SQRT (J)

```

The next execution function which can be performed while the speed is at STOP is the setting and displaying of HLL program variables at the bottom of the screen, (Figure 4). If WEIGHT is a single valued program variable and one writes

```

WEIGHT = 65.25

```

at the bottom of the screen, then WEIGHT is set to that value. If one writes

```

HEIGHT (12,3) =

```

without supplying a value, then GRAPE fills in the present value of HEIGHT_{12,3} to its full computer precision.

```

HEIGHT (12,3) = .32750+-04

```

One may wonder how the system knows that an individual has stopped writing at an equal sign and is therefore requesting the value of a variable rather than supplying one. While teletype input often uses a special character, such as carriage-return, to indicate the end of input, graphical input tablet character recognition typically uses timing information. A significant delay indicates that input is complete. But nothing happens to the user who inadvertently delays before writing his new value. As will be seen in 2.4, overwriting is the standard way of changing information which is on the screen.

With regard to the example above, GRAPE could understand the mathematical notation for subscripts, but it is wiser to have the variable look just as it does in the programming language, and subscript notation is not used in FORTRAN. All the current HLL have forced a linearity on the basic two dimensionality of mathematical statements. ALGOL (and PL/I) have the additional inherent two dimensionality of statements containing strings of ELSE clauses. "If X_1 is true do Y_1 , ELSE if X_2 is true do Y_2 , ELSE..." GRAPE is perfectly well suited for a statement oriented two dimensional higher level language, and so is the graphical input tablet, as Andersen (ref. 8) has shown.

After program execution is continued these variables remain displayed at the bottom of the screen and their values are changed on the screen as execution of the program changes them. If a GRAPE user were looking for the place in a program where the value of a variable changed from a small integer to a large one, he could display the variable on the screen and then run the program at a high speed until the variable changed value. Displaying and setting the values of entire arrays, rather than variables with a single value, involves the same problems that display output during execution involves, and it is treated in 2.5.

The final execution function is the ability of the user to condense source program code. Condensing is used to remove from the screen code which is no longer interesting. Drawing a circle around a set of contiguous line numbers reduces all the encircled lines to a single line on the screen. When lines of code are condensed, the first line remains on the screen with a condense square, \square , in the special character column.

The code

```
123      I = 14
124      J = 15
125      K = 16
```

126 L = 17

127 M = 18

when the user draws

123 I = 14

124 J = 15

125 K = 16

126 L = 17

127 M = 18

becomes

123 I = 14

124 □ J = 15

127 M = 18

Two important purposes are served by condensing. First, code which the user is not interested in seeing displayed is removed from the screen, though not from the program, thus increasing the amount of useful information which can be displayed at one time. Second, the code which is removed from the screen is executed by the computer at full computer speed, an increase of up to 10000 percent over even the fastest execution that still involves changes to the display screen.

During execution the condense square brightens while execution is taking place in the lines which are not on the screen. As long as the code which is hidden from the screen is not very time consuming, there is no perceptible delay from the display of execution of the line with the condense character to the display of execution of the next line. In the above example, three execution cycles would be needed to execute the program with the condense block. The second cycle would indicate how line 124 was to be executed, but lines 125 and 126 would also be executed immediately following this cycle.

Suppose that there were a fifty line program in which appeared the twelve lines of Figure 7. Display of this section of code takes over 25 percent of the screen's capability of forty lines. Worse yet, it involves the execution of nearly five thousand separate statements, some of which take more than one cycle to display. Even with no delay at all between execution cycles,

| | | |
|----|----|-----------------------|
| 14 | | INTEGER A,B,C |
| 15 | C | INITIALIZE ALL ARRAYS |
| 16 | | DO 20 I = 1,100 |
| 17 | | A(I) = 0 |
| 18 | | B(I) = 0 |
| 19 | | C(I) = 0 |
| 20 | | DO 15 J = 1,10 |
| 21 | | D(I,J) = 1.0 |
| 22 | | E(I,J) = -1.0 |
| 23 | 15 | CONTINUE |
| 24 | 20 | CONTINUE |
| 25 | | DO 30 RADIUS = 1,50,2 |

Figure 7.- Condensing A Program Loop


this section of code could take over a minute (100 times as long as a "normal" 10000 statements per second) to execute and display.

By drawing the indicated circle around the line numbers, the code is reduced to

```
14          INTEGER A,B,C
15  C  □ INITIALIZE ALL ARRAYS
25          DØ 30 RADIUS = 1, 50, 2
```

Now the entire executable part of the fifty line program fits on the display screen, and the whole array initialization takes place in one execution cycle. Between the single execution cycle of the comment statement and the single execution cycle of the DO statement, the arrays are initialized. However, if the code hidden from view involved a million calculations, then it would take longer than a brief moment to execute that hidden code. For the time that the condensed code was operating the computer, the condense line would be intensified as

```
15  C  □ INITIALIZE ALL ARRAYS
```

If execution reaches a halt, e.g., a PAUSE statement, inside the condensed code, the message at the top of the screen changes from GO to STOP, just as if the statement were displayed. The user could continue execution, just as if the statement were displayed, by moving the speed control to STOP and then away from it. If he did not know why execution had stopped, he could pull the speed to STOP and then make the hidden code appear by crossing out the condense square, that is, by "erasing" or scrubbing over it, as in . (Breaklines also are removed by scrubbing over them.) Because of the variability in execution time, when an input/output statement executes inside a condensed block the code below the condense line drops down, the input/output statement appears, and it is executed visibly.

Statements which transfer control to the middle of a condense block, e.g., GO TO n where statement n has been condensed, work as might be expected; the line which is intensified, and brought onto the screen if it is not already there, is the condense line at the beginning of the block. As that implies, condense blocks remain condensed even when the whole section of the program is off the screen. There can be any number of condense blocks in a program. Furthermore, condense blocks can include other condense blocks.

| | |
|----|-------|
| 10 | I = 1 |
| 11 | J = 2 |
| 12 | K = 3 |
| 13 | L = 4 |
| 14 | M = 5 |

becomes

| | |
|----|-----------------|
| 10 | I = 1 |
| 11 | \square J = 2 |
| 14 | M = 5 |

and

| | |
|----|-----------------|
| 10 | I = 1 |
| 11 | \square J = 2 |
| 14 | M = 5 |

becomes

| | |
|----|-----------------|
| 10 | \square I = 1 |
|----|-----------------|

In this case

| | |
|----|----------------------|
| 10 | $\sim \square$ I = 1 |
|----|----------------------|

becomes

| | |
|----|-----------------|
| 10 | I = 1 |
| 11 | \square J = 2 |
| 14 | M = 5 |

However, if the first line in a new condense block is already a condense line, only one condense block results.

| | |
|----|-----------------|
| 10 | I = 1 |
| 11 | \square J = 2 |
| 14 | M = 5 |

becomes

```
10      I = 1
```

```
11      □J = 2
```

and if the square is crossed out

```
10      I = 1
```

```
11      J = 2
```

```
12      K = 3
```

```
13      L = 4
```

```
14      M = 5
```

Two final remarks about condensing must be made. Subroutine calls initially appear as condense lines, and the execution of subroutines takes place in two cycles as indicated in 2.2.

```
145      □CALL NEWTON (OLDVAL, 5.3, LOGVAR, NEWVAL, ITERNO)
```

If one wishes to see the subroutine execute, one crosses out the condense square and the subroutine code appears under the call. The execution of the subroutine occurs in normal GRAPE fashion between the TO and FROM cycles of the subroutine call. Subroutines may themselves contain condense blocks, such as other subroutine calls.

Single statements can be condensed by drawing a circle around the single line number. The effect of condensing a single statement is that it executes in one cycle, and statements containing complicated expressions may warrant such condensing. If one condensed the following statement by drawing a circle around the 160

```
160      A(K+3,J) = J*(SQRT(X(K-1)) + X(K-2))
```

the single cycle of execution might appear as

```
160      □A(K+3,J) = J*(SQRT(X(K-1)) + X(K-2))
```

```
A( 10,4) = 32
```

If only that one line is condensed the next line on the display screen would have the line number 161, hence the user would know that the condense block was just a single line.

2.4 Editing Functions

2.4.1 Changing Code. - Changing the actual source program code can be accomplished whenever the speed control is in the STOP position. GRAPE itself is not an editor. Rather, it is the capabilities of the graphical input tablet with sufficient supporting software that make the editing feature of GRAPE so powerful. Editing a page of text with a tablet is even easier than annotating a page of text with a pencil for someone else to edit. To change letters or words in a line, one simply writes over them. To insert letters or words in a line, one draws a caret and inserts the information. To insert a new line draw a > between the line numbers where the new line is to go; the code on the screen separates to allow writing of the new line. It has been noted that breaklines and condense blocks can be removed by crossing them out. Letters or words can be removed the same way. An entire line can be deleted by crossing out the line number.* Some examples follow.

If the code looks like this

```
15          GØ  TØ  20
16    10      I = J + 2
17          K = 14 + M/2
18          L = 26
```

and the user crosses out the 17

```
15          GØ  TØ  20
16    10      I = J + 2
17          K = 14 + M/2
18          L = 26
```

the code becomes

```
15          GØ  TØ  20
16    10      I = J + 2
17          L = 26
```

*I have taken these ideas from Project GRAIL at Rand Corporation, one of the sophisticated users of graphical tablets. They are the designers of the Rand Tablet, the first commercially available version of the graphical input tablet.

Line 16 could be changed to read 10 I = J+4-M by just writing
the new code on top of the old

```
16  10      I = J + 4 - M
```

It is not necessary to indicate in any way that changes are about to be made, or indicate which line is to be edited. One simply writes the changes. If in the above example one wished to add a line between lines 15 and 16 one would use a >

```
15      GØ  TØ  20
>
16  10      I = J + 2
17      L = 26
```

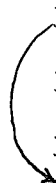
The code becomes

```
15      GØ  TØ  20
16
17  10      I = J + 2
18      L = 26
```

and one can write whatever code he wants on line 16. These features have all been used by Project GRAIL. They are the common editing features of ^ , > , ~~~~~~~~ and overwriting. One could get the value of some function of the program variables by inserting an arithmetic statement anywhere on the screen, executing it, then deleting it. Besides the statement itself, all one writes is > , ~~~~~~~~ , ~~~~~~~~.

A fifth common technique which is used in GRAPE is moving a line by drawing an arrow from it to where it should go.

```
14      A = 1
15      D = 4
16      B = 2
17      C = 3
18      E = 5
```

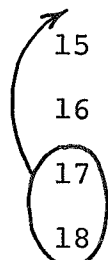


becomes

```
14      A = 1
15      B = 2
16      C = 3
17      D = 4
18      E = 5
```

One can also move a block of lines easily

```
14      A = 1
15      D = 4
16      E = 5
17      B = 2
18      C = 3
```



The circle condenses the lines, the arrow moves them. One may then cross out the condense square

```
14      A = 1
15      □ B = 2
16      D = 4
17      E = 5
```

These five functions are all simple and are all common editing techniques. The thing to realize is that indicating the changes actually produces the changes.

There are many other features for changing code which should be found in a good general purpose editor, such as locate-by-context or change-everywhere. But complex code modification belongs in the editor, not in GRAPE. See, for example, reference 9 Section AH.3. The five features above are all that are needed for easy line by line editing. It is not unreasonable to suppose that, with a good operating system, the GRAPE user will find it simple to use the available general purpose editor on his HLL program.

2.4.2 Checking Code. - Before touching on other types of editing, let us state a point that is probably already clear about line numbers: they are not permanent. The lines in the main program are numbered in

order. Any deletion or addition changes the line numbers which follow. Line numbers serve two functions in GRAPE. The first is to mark the lines that are on the display screen. Markings on the left of the screen provide a convenient place to indicate insertion of a line, deletion of a line, condensing of lines, particularly since HLL source statements do not have to be aligned vertically. For this a mark such as \odot would serve. But actual line numbers can also be used to roughly identify sections of a very long program. One may remember that a statement is in the vicinity of line 200. However, permanent identification of a line number with a particular statement does not provide additional capability and so the machinery necessary to provide it has not been included in GRAPE.

We have already seen how the values of single-valued variables can be displayed at the bottom of the screen. To set the value of a variable write

```
VARIAB = 38.6
```

To display the present value write

```
VARIAB =
```

Furthermore, if the speed is STOP, the value of a variable being displayed can be changed by writing the new value over the old. For example,

```
VARIAB = $$
```

This will reset VARIAB to 11. The display of a variable can be removed from the screen by crossing it out. Similarly, a new variable can be displayed in place of one already there by writing the new name over the old.

```
A = 29.1
```

The question arises as to how many variables can be displayed at one time at the bottom of the screen. The answer shows the flexibility of GRAPE and computer driven displays. When GRAPE begins operating, only one line is reserved for the display of variables. The rest of the screen, except for the message area at the top, is available for the HLL program. This line is divided into fourths as in Figure 8. As long as three or fewer variables are being displayed, just the one line is taken. As soon as the display of a fourth variable is requested, the bottom line of the HLL program disappears from the screen and a second line of four display locations is added as in Figure 9. After four more displayed variables, this action is repeated. Adding of display locations for variables can continue until the HLL program has been completely eliminated from the screen. Conversely, as displayed variables are crossed out, display locations disappear and program code reappears.

| | | | |
|-------|-------|-------|--|
| 123 | | | |
| 124 | | | |
| 125 | | | |
| 126 | | | |
| A = 1 | B = 2 | C = 3 | |

Figure 8. Three Displayed Variables

| | | | |
|-------|-------|-------|-------|
| 123 | | | |
| 124 | | | |
| 125 | | | |
| | | | |
| A = 1 | B = 2 | C = 3 | D = 4 |

Figure 9. Four Displayed Variables

Certain cross reference information occasionally found in the output of a good compiler can be used to great benefit in GRAPE. Writing 120? at the bottom of the screen (while the speed is at STOP) illuminates the line numbers of all statements that could transfer control to statement 120 (Figure 11). Writing X? illuminates the line numbers of all statements which contain the variable X (Figure 12). These references illuminate on all pages of the program, so one can ROLL through the program to find all the references. Restarting program execution removes the illumination as does crossing out the request.

The GRAPE interpretive compiler performs considerable error checking at execution time, such as checking the range of subscripts. This information is almost always useful when one is trying to analyze a program, especially a faulty one. The extra time taken is relatively small in most interpretive compilers and is meaningless in a system such as GRAPE which has a built in delay. When an error condition is found, the system displays the diagnostic and halts as shown in Figure 10. Moving the speed lever to and from STOP will cause the compiler to make a best guess and continue execution, if any guess as to meaning is possible. The user need not memorize what the best guess will be in any given situation, since proceeding with the execution will give him that information. If he does not like the guess he can take appropriate action. In condensed statements, which are "incrementally" compiled and run at full machine speed, no such checking is done, and so there is a natural way to allow the user to write "illegal" statements which will perform correctly.* He does just what he did to eliminate the tedious display of an expression evaluation; he condenses the single statement.

Another feature which is useful in program analysis is the illumination of code which has never been executed. When the speed is at STOP, one of the messages at the top of the screen is UNUSED. (Refer to User's Manual.) Touching it with the pen causes the brightening of the line numbers of all statements which have not been executed since the last time the program was started at the beginning. Comment statements and certain other non-executable statements are not illuminated by UNUSED. Performing any other GRAPE function, except ROLL, removes that illumination. Figure 10 shows part of a program which was started at line 1 and allowed to run until it halted. Figure 11 shows the results when the speed lever was moved to STOP and 100? was written at the bottom of the screen. When JSET? was written on top of the earlier request, Figure 12 resulted. When JSET? was crossed out and UNUSED was touched, Figure 13 resulted.

*Statements which violate some rule of the language, but which the user knows will run correctly. Also included would be error statements whose default executions are acceptable.

| | | | |
|---------------------|-----|----------------------------------|--|
| STOP | | | |
| 1 | | DIMENSION TABLE(20,30) | |
| 2 | | INDEX = 2 | |
| 3 | 100 | GO TO (200,300,400), INDEX | |
| 4 | C | | |
| 5 | 200 | JSET = 6 | |
| 6 | | INDEX = 2 | |
| 7 | | CALL FILLER (HEIGHT) | |
| 8 | | GO TO 100 | |
| 9 | C | | |
| 10 | 300 | INDEX = 3 | |
| 11 | | CALL FILLER (WEIGHT) | |
| 12 | | GO TO 100 | |
| 13 | C | | |
| 14 | 400 | TABLE(JSET,10) = HEIGHT * WEIGHT | |
| JSET IS NOT DEFINED | | | |
| 15 | | L = 3 | |
| 16 | | M = 20 | |
| | | | |

Figure 10.- Error Message

| STOP | RESTART | SAVE | TRASH | UNUSED | ↑ROLL↓ |
|---------------------|---------|-----------------------------------|-------|--------|--------|
| 1 | | DIMENSION TABLE (20,30) | | | |
| 2 | | INDEX = 2 | | | |
| 3 | 100 | GO TO (200,300,400), INDEX | | | |
| 4 | C | | | | |
| 5 | 200 | JSET = 6 | | | |
| 6 | | INDEX = 2 | | | |
| 7 | | CALL FILLER (HEIGHT) | | | |
| 8 | | GO TO 100 | | | |
| 9 | C | | | | |
| 10 | 300 | INDEX = 3 | | | |
| 11 | | CALL FILLER (WEIGHT) | | | |
| 12 | | GO TO 100 | | | |
| 13 | C | | | | |
| 14 | 400 | TABLE (JSET,10) = HEIGHT * WEIGHT | | | |
| JSET IS NOT DEFINED | | | | | |
| 15 | | L = 3 | | | |
| 16 | | M = 20 | | | |
| 100? | | | | | |

| STOP | RESTART | SAVE | TRASH | UNUSED | ↑ROLL↓ |
|---------------------|---------|------|-------|--------|--------|
| 1 | | | | | |
| 2 | | | | | |
| 3 | 100 | | | | |
| 4 | C | | | | |
| 5 | 200 | | | | |
| 6 | | | | | |
| 7 | | | | | |
| 8 | | | | | |
| 9 | C | | | | |
| 10 | 300 | | | | |
| 11 | | | | | |
| 12 | | | | | |
| 13 | C | | | | |
| 14 | 400 | | | | |
| JSET IS NOT DEFINED | | | | | |
| 15 | | | | | |
| 16 | | | | | |
| JSET ? | | | | | |

Figure 12.- Variable Cross Reference

The final message displayed at the top of the screen when execution is halted is used for the last editing feature. Users are bound to make mistakes when they edit code. They will delete the wrong line or will make a change and only later realize the change should not have been made. The operating system under which GRAPE runs should provide facilities for hard copy listing of files. Since SAVE does not modify the running HLL program, one could SAVE, create the listing, and resume GRAPE operations. However, GRAPE has an easier feature to permit recovery from an incorrect correction. Teletype users have the teletype paper to indicate the changes that have been made. Likewise, GRAPE has a first-in, first-out TRASH file in which is stored the last one or two hundred lines of modifications that were made to the program. This includes deletions and changes. In addition, the TRASH file has the line numbers, in order, of the last hundred non-condensed statements which were executed.

To look at this file one touches the message TRASH with the stylus and the most recent thirty lines of program changes appear on the screen. At the top of the screen the status message at the left still reads STOP, and the only other message is

↑ROLL↓

(See Figure 14 or the User's Manual.) One can roll through the trash file just as one rolls through a program. The way to terminate display of the trash file is to continue with program execution. This could be done by moving the speed lever away from STOP but it can also be done by pressing the lever while it remains in the STOP position. Since execution will start at the beginning of the statement during which execution was halted, this single cycle execution will have no effect on the program operation. However, it will restore the program to the screen.

2.5 Special Situations

There are four situations which demand special attention. They are (1) time dependent programs, (2) programs with display output, (3) display of multi-valued variables, (4) extra long programs.

(1) In general, time dependent programs cannot now be run under GRAPE. GRAPE does not work as if the basic cycle time of the computer were being slowed down. Even though GRAPE could conceivably count the computer cycles necessary for the execution of a given source program statement and could simulate interrupts, the user would not be seeing what this particular user was really interested in seeing, namely the machine status during the interrupt. However, it is possible to use GRAPE to test interrupt driven programs in which the timing itself is not so critical. The user performs the interrupt by halting execution and starting it at the beginning of the interrupt handline routine. The insertion and deletion of program statements to create certain conditions is simple with GRAPE. If one knows that problems arise when the value of an index variable gets to 1000, he can insert

STOP

↑ ROLL ↓

10 IF (I.GT.J) GO TO 18

32 DO 60 K=1,100

33 MAT1(K) = 0

34 MAT2(K) = 0

10 IF (I.GE.J) GO TO 18

. . .

. . .

. . .

9 IF (I.GT.J) GO TO 18

50 51 52 53 50 51 52 53 50

51 52 53 50 51 106 107 108 112

MOST RECENT LINES

Figure 14. Trash File

IF (I.EQ.1000) PAUSE

and then run the program at maximum speed. If one wishes to branch to an interrupt routine and let GRAPE remember where to branch back to, one can insert a subroutine call to the routine. Thus, almost the entire portion of, say, a time-sharing system could be analyzed although certain time critical contingencies would not be tested.

(2) The fact has already been discussed that display programs which slowly create pictures, perhaps even light pen sensitive pictures, are difficult to handle on the GRAPE display. However, there is no reason why a graphics user should not have two display consoles, one for GRAPE and one for the pictures. If two displays are not available, there is an alternative solution. Providing the program is not an interactive one, that is if no graphical input is involved, the user could alternate looks at the picture with execution of his program. The computer operating system would be responsible for saving and restarting both GRAPE and the partially created picture. A split screen technique with GRAPE on one side of the display face and the picture on the other side is also a possibility, but for most displays considerable additional programming effort would be involved.

There is a programmed use of the display which GRAPE does handle: the use of the display to simulate character output on a printer or a tape. Typical of this is the statement

```
WRITE (DEVICE, 100) ARRAY, VARIAB, I, J, K
```

where DEVICE will be a standard output device when this source program is operating for production but which is the display while the program is being tested.

The method of display is exactly the same as the method for display on the trash file. The entire display screen, except for the top inch or so, is used to perform the output requested in the source statement. At the top left of the screen is STOP. At the top middle is \uparrow ROLL \downarrow . ROLL is meaningful only if the source statement requests more output than can be fit on the screen at one time. Touching the down arrow with the stylus puts the next lines of display output on the screen. To terminate the output and continue with the execution of the program, one moves the speed lever to and away from STOP. Figure 15 is a typical display.

(3) When the speed is set to STOP, one can request the display of variables which are not single valued (e.g., arrays, trees, switches). The display of such variables takes place exactly as with other requested display output; the source program code is taken off the screen and, except for the message space at the top, the screen is used for the display of the variable. The status message is STOP (since the speed control is at STOP); the only other message is \uparrow ROLL \downarrow . The way the variable is displayed will depend on its data structure.

STOP

↑ ROLL ↓

THIS IS DISPLAY OUTPUT
WHICH COULD ALSO BE PRINTED
ON A TELETYPE.

HEIGHT = 0.0

WEIGHT = 372648

Figure 15. Display Output

In FORTRAN the only allowable data structure is the array. For a 1-dimensional array, the array is displayed, to its full computer precision, with about seven entries per row. If ARRAY were a 1-dimensional array of length 24, writing "ARRAY=" at the bottom of the screen would produce the display of Figure 16. If ARRAY were four rows by three columns Figure 17 would result. If ARRAY were 50 rows by 9 columns Figure 18 would result.

In the last example, \uparrow ROLL \downarrow can be used to display the other values of the subscripted variable, ARRAY. To terminate the display and bring back the program code, either move the speed lever away from STOP or simply press the lever for single cycle execution.

(4) Finally, what of the very long programs? Can GRAPE be used to analyze a program which after standard compilation takes several minutes to run, since displaying the execution of all lines may take thousands of times as long? The answer is that the longer the program, the more useful GRAPE becomes. With a short program one can often memorize the entire program, and any clues at all will be sufficient to explain abnormal behavior. One could even tolerate an octal dump if the program were short enough. Likewise, one is willing to study another person's program if it is reasonably short.

It is the long program, with dozens or hundreds of internal variables and numerous loops and convolutions, that is most difficult to debug or modify. And it is most difficult precisely because of the amount of information which is inaccessible to the user both while the program is (incorrectly) executing and while the user is looking at a listing. The problem of time for execution is not serious since a well implemented GRAPE uses an incremental compiler and produces real machine code for condensed blocks. This mode of execution takes only slightly longer than full machine speed execution. Furthermore, one would not run every program statement at delayed speed. The initialization loop in Figure 7 would be run perhaps twice before it was condensed. All the rest of the iterations would run at full machine speed.

A savings of many, many hours of human time is certainly worth some extra computer time. And because of the reduced number of passes necessary to debug a program with GRAPE, combined with a time-sharing system which lets other users operate during the interstatement delays, even the computer time may be reduced.

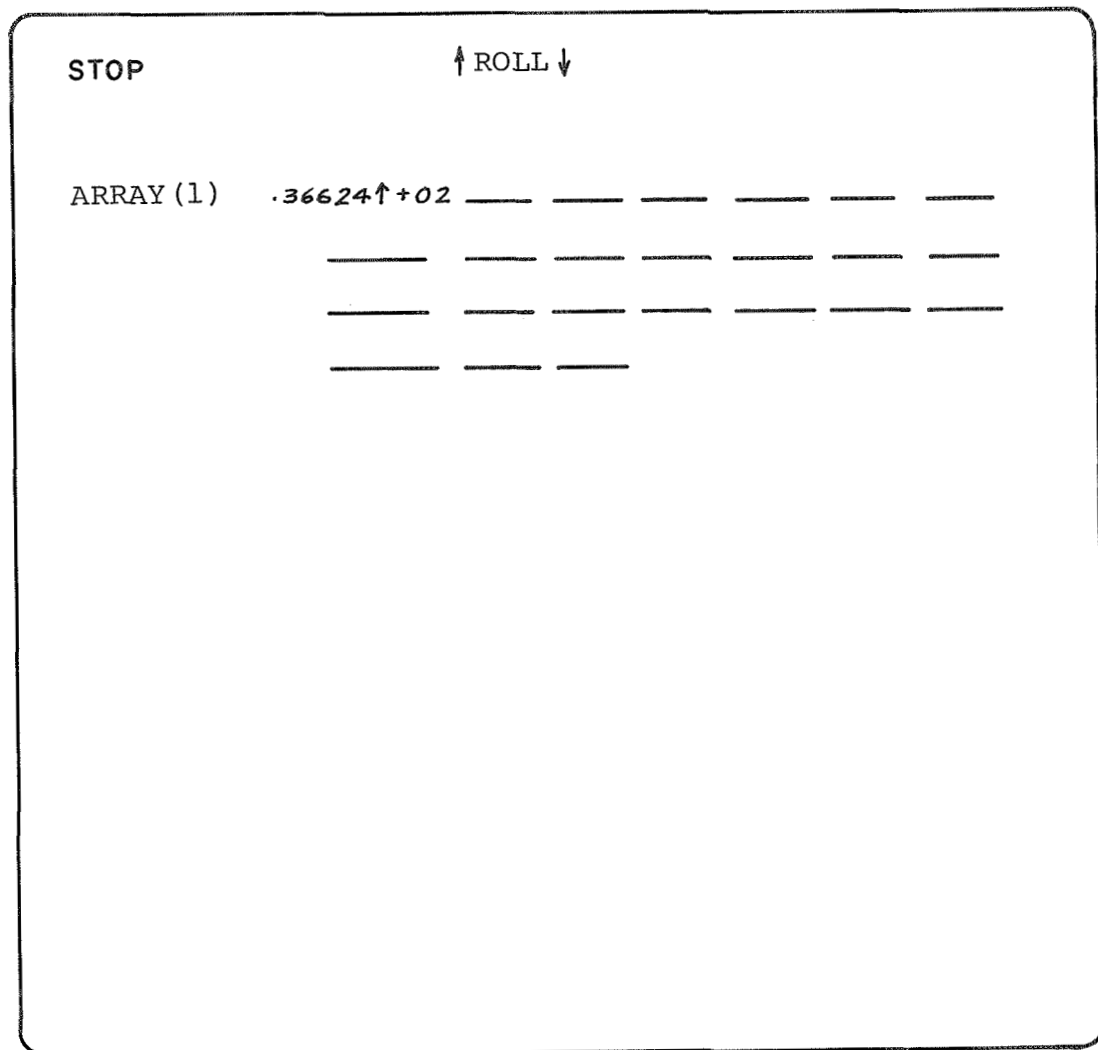


Figure 16. ARRAY (24)

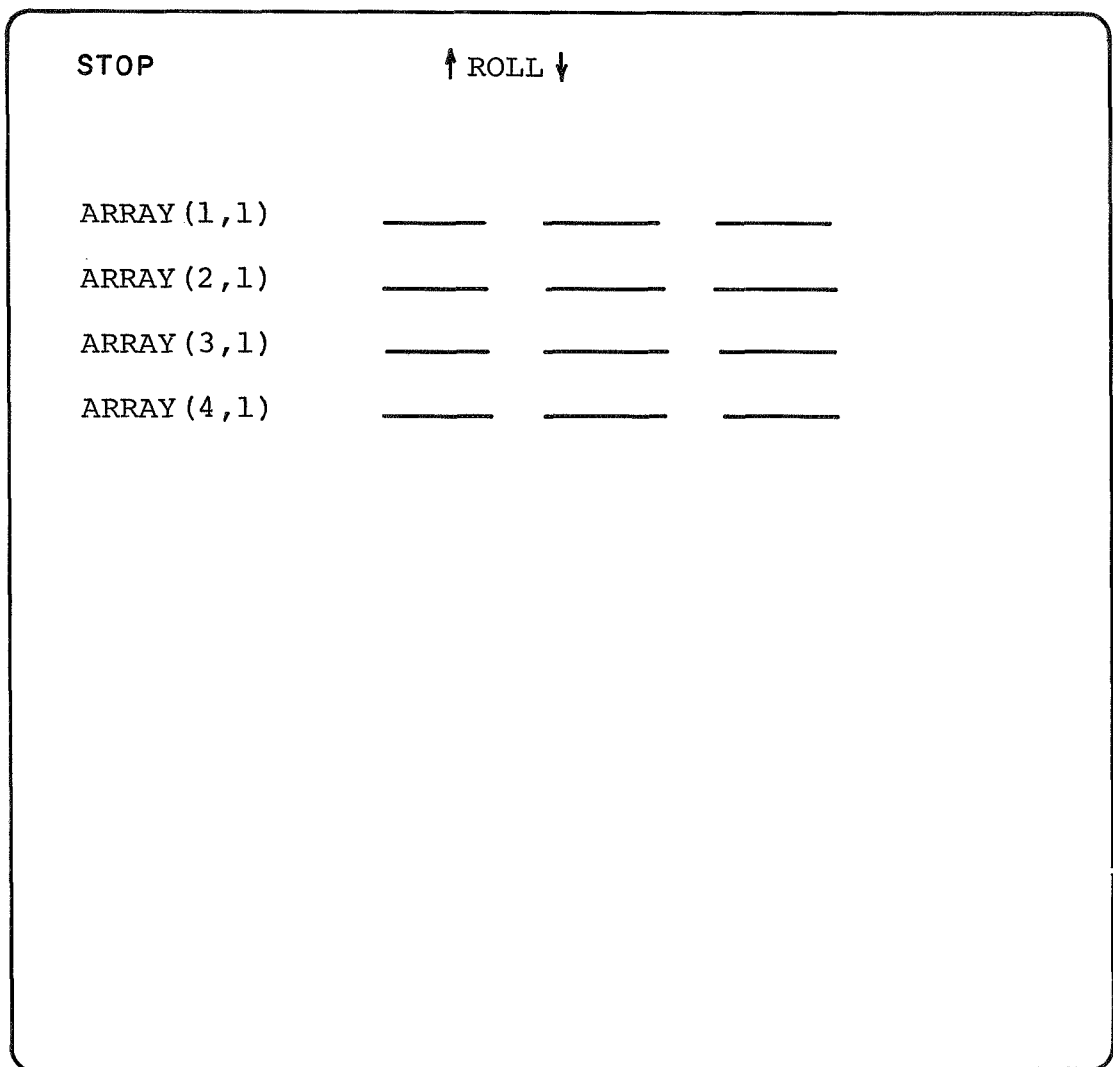


Figure 17. ARRAY (4,3)

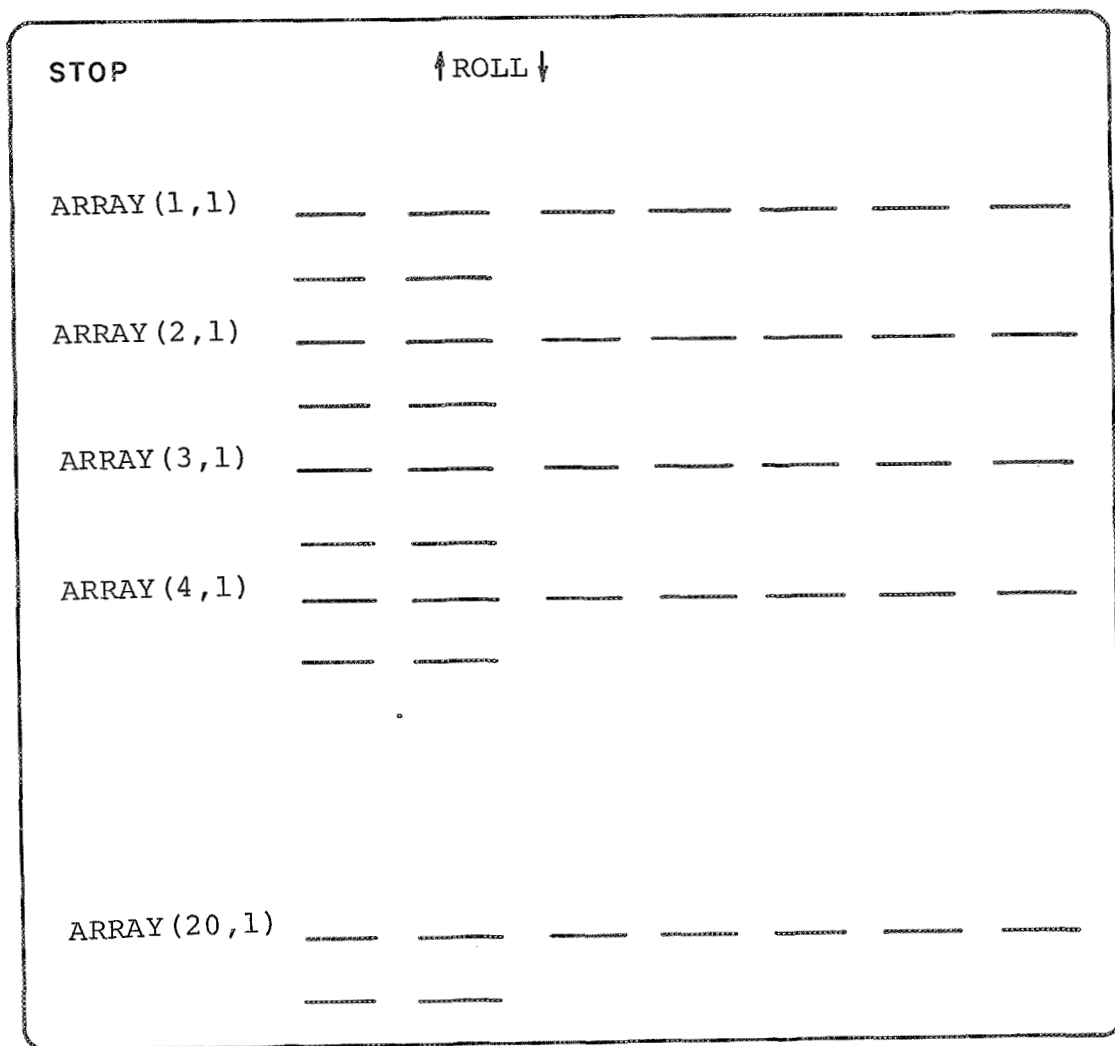


Figure 18. ARRAY(50,9)

2.6 USER'S MANUAL

| <u>SITUATION</u> | <u>MESSAGES</u> |
|--|--|
| Program execution | GO |
| PAUSE, STOP, END Breakline, Diagnostic | STOP |
| Display output, TRASH Multi-valued variable | STOP ↑ROLL↓ |
| Speed set to STOP | STOP RESTART SAVE TRASH UNUSED ↑ROLL↓ |

RESTART initializes program variables

SAVE saves the program and returns to the operating system

TRASH displays recent code changes and an execution trail

UNUSED illuminates never executed code

ROLL rolls the program

SPECIAL CHARACTERS

| | |
|-------------------------------|---------------------------------|
| —→ Start execution here | > Insert a line |
| ┌ Breakline (stops execution) | ^ Insert text |
| □ Condense to one line | ~~~~ Erase a line or a function |
| | (Move a line |

BOTTOM OF THE SCREEN

| | |
|-----------|---|
| X = | requests the value of X |
| X = 143.2 | sets the value of X |
| X? | illuminates references to X |
| 120? | illuminates references to statement 120 |

III GRAPE PRACTICE

3.1 Novel Hardware And Software Building Blocks

3.1.1 Required. - GRAPE draws on a variety of new and experimental techniques in computing. Although all of them exist in the computing world, several of them are still at the stage where implementation is a separate research effort of itself. Thus a full implementation of GRAPE in 1968 would have been a several man effort. Fortunately, enough hardware resources were available so that a GRAPE implementation could be produced which retains many of the novel ideas given here. This version was written to test these novel features and to demonstrate the usefulness of the GRAPE approach. In particular, all the source program execution and the execution function features are retained, although some flexibility has been lost. Several of the editing functions do not exist. Throughout this discussion, "source program" refers to the user program being analyzed under GRAPE, a "statement" is an instruction in the source program, and a "line" is a line on the screen. For example, the source program statement "80 GØ TØ 90" may be line 15 on the display.

The GRAPE hardware and software building blocks that were novel in 1968 are as follows:

1. A computer driven refresh cathode ray tube with the capability of drawing several hundred characters and with point drawing capability.
2. A computer driven graphical input tablet.
3. Software to control the display.
4. Software to control the tablet (and to associate it with the display).
5. Character recognition software to interpret information from the tablet.
6. An interpretive compiler for the higher level language which will produce code that can be handled by the execution phase.
7. The execution phase of GRAPE, which will not only perform the actions required by the source language statements but will also take care of updating the information on the display.
8. The editing phase of GRAPE which will allow both editing of the source language statements and inputting of execution instructions (such as " □ ").

3.1.2 Provided. - The computer on which GRAPE is implemented is a Honeywell DDP-516; a 16-bit, 1-microsecond machine with a multiplexed input/output channel. The mass storage device is a standard disc drive similar to the IBM 2311. In a later section, the transportability of GRAPE to other machines will be discussed. Each of the building blocks listed above was handled as follows:

1. The display is made by IDI (Information Display Incorporated). It has character, vector, and point generators, and a light pen. It does not have subroutines, jumps, or breaks. More than one display station can be run from the display generator. At present it is driving three IDI display stations as described above and one ITT 3-color display. The color display station does not have a hardware character generator, and no attempt has been made to put software generated characters on its screen. In Section V there are comments on adding other hardware capabilities to the GRAPE system.

It may be argued that excellent equipment should not be a prerequisite for this type of research. Such is true and the display had more capability than was needed. However, lack of a hardware character generator would be a serious deficiency. This is not so much because of the extra programming involved in writing a software character generator but because of the reduction in material which could be displayed on the screen. When the flicker on a display is severe, textual information is much harder to comprehend than picture information. Since software generated characters would take 5 to 10 times as much display time to put on the screen (for this particular display), and since the flicker is already slightly noticeable, considerably less material would be visible to the user at one time. But a fundamental concept of GRAPE is to give the user a window to his program, and there is a critical size below which the window is no window at all. Seeing ten or even twenty lines of a higher level language does not give the user the feeling that he can see almost everything he wants to. As a result, hardware character generation is a GRAPE necessity.

2. Both a BBN Grafacon 1010A and a Sylvania Data Tablet 1 were at the installation and this availability had a great impact on the basic design of GRAPE. It is the graphical input tablet that permits editing directly on the source program itself and that allows direct input of the execution functions. A listing of a program or a display of a non-executing program is two dimensional, and the tablet permits corresponding two-dimensional input. All the input problems of addressing and location which occur if the input device is one dimensional, such as a teletype, disappear when the tablet is used. However, for reasons discussed in 5 below, the tablets were not used in the present implementation.

3. There are several levels of software operations which must exist in some form for the display. First is the basic function of refreshing the display at regular intervals and servicing the interrupts from the display. Second is performing user requested control

functions, such as disabling the light pen. Third is permitting the user to change the structure or contents of the display code buffer.

The complexity of these functions will depend on the hardware characteristics of both the computer and the display. In the present case, not only did no software exist for the display, but the economies of Section I applied, and it was necessary that the display run in a time-shared environment. Furthermore, it was intended that there be multiple displays running simultaneously in this environment. (The author was involved in the design of the time-sharing system and was able to guarantee that the system was conducive to the use of displays.)* The precise implementation of the various levels of software is not of interest here, but the incorporation of computer driven tablets in a time shared environment is somewhat unusual, and the results of this effort are described in Section IV.

4. Software was developed according to the design in Section IV to control the graphical input tablet. The software accepts data from the tablet, delivers the data to the user (probably the character recognizer), and associates a tablet with a display for the temporary trace. But the graphical input tablet has not been used in the present implementation of GRAPE.

5. Character recognition is not a feature of the displayed execution of GRAPE. It is, however, important in the "naturalness" and total environment supplied by GRAPE. While many people compose papers at a typewriter, few prefer a typewriter to a pencil in order to indicate the errors on a page of text. Character recognition is the capability of the computer (via software) to recognize input from a graphical tablet as being certain letters (ref. 10). Those who have seen editing being done on a graphical input tablet with a good character recognizer and a good editing program already appreciate the usefulness of this tool. Those who are not familiar with this technique may just consider how often they have made penciled corrections on a sheet of paper and wished that the corrected version would magically appear. Certainly the naive computer user would insist and the experienced user would prefer that the tablet be available to him.

Because of continuing hardware difficulties and delays connecting the tablets to the computer and because the character recognition is not vital to the demonstration of GRAPE, use of the tablet was foregone and the teletype was used instead. It was desirable, however, to use graphical methods rather than teletype input for the man-machine interaction which occurs during the source program execution. Therefore, the speed of execution of the source program is controlled with the light pen. Appropriate light pen software was already written (by this author) as part of the display support system.

6. Some sort of compilation of the HLL program is required which permits analysis of the source language statement and display of the temporary results according to the original HLL statement, for

*A description of the graphics software can be found in Section IV.B of the NASA/ERC Computer Research Laboratory Systems Programming Manual.

there is not enough information in the machine language to produce the displays desired. One must retain not only the character string of the source code but also information about the statement's meaning. This can be done by interpretive compiling, that is by recompiling each statement every time it is executed, or by incremental compiling, that is by compiling each statement once and storing it in some intermediate code. This latter makes the statement easy to execute but retains the necessary information to make it properly displayable. More sophisticated incremental compiling which would bind blocks of executable code together could be used for sections of code which are not to be displayed. Notice that it must be possible to start execution of the source program anywhere, even following code which is incorrectly written. Such modes of compilation mean the existence of a properly constructed compiler. No such compiler existed for the DDP-516, therefore, this author wrote one.

Writing a topnotch interpretive or incremental compiler is not a straightforward task because the ideas are not so well understood as with normal compilers. To simplify the compiler's job, the source language was simplified. In particular, some FORTRAN statements were left out, the formats of permissible statements are rigidly defined, and little error checking is done. The syntax rules for the FORTRAN version used in this implementation will be given in 3.2. Section 3.3 contains a description of the data bases and of the compiler.

7. The source program execution and the execution functions work essentially as described earlier. Some reduction has been made in the sophistication of the material displayed, but the results of every statement are still clearly indicated. The speed control works by means of a light pen visible speed line on the display face.

All the execution functions are implemented. They are invoked via a keyboard rather than a tablet, but they still appear on the display.*

8. The editing functions in the implemented version of GRAPE are different from what was described above. First, all editing

*I write keyboard rather than teletype because GRAPE's reaction to the input is always on the screen, not on the teletype paper. Thus, even though it is a teletype that is active, only the keyboard part is actually used.

is done via keyboard. (As with execution functions GRAPE responds on the display screen.) Second, TRASH is not implemented nor is UNUSED and associated functions. These features do belong in any production version of GRAPE, especially for use with long HLL programs.

3.2 Implemented GRAPE -- User Side

3.2.1 Basics. - The design philosophy of this implementation of GRAPE is the same as that of the ultimate GRAPE. What makes this version different at the conceptual level is the change in hardware. The user has a display, a light pen, and a teletype.

The light pen is used only for controlling the speed of execution of the source program. Touching the speed line with the light pen sets the speed of execution; the top is the fast end, the bottom is the slow end, the very bottom is STOP. The section of the line corresponding to the present speed is brighter than the rest of the line. Similarly the word STOP serves as the status message; it is displayed more brightly when the program is not executing.

The teletype is used to input commands to GRAPE (commands to perform the execution and editing functions) and to input changes to the source program code. A command is composed of a command code and one or two operands which are generally line numbers. For example to delete line 14 one types DELETE 14.* To condense lines 20 through 25 one types CONDENSE 20 25. Here one is issuing commands to a system to perform a function, whereas with the graphical input tablet, one would perform the function directly. However, all functions have an immediate visible result so the intruding system is existent but not mysterious.

The display screen setup is shown in Figure 19. The permissible source program statements are all short, so the displayed variables were moved to the right side of the screen, leaving room for additional lines of code. STOP serves as the status message. No other messages appear since functions are initiated at the keyboard.

3.2.2 Source Program Execution. - Source program execution takes place in a manner similar to the execution described in 2.2. The program is executed one statement at a time, and appropriate results are displayed. When control transfers to a statement which is not presently on the display, a new page of source program code headed by that statement is put on the display. The line number of the presentl executing statement is brightened as is the section of the speed line corresponding to the present execution speed.** Displayed variables

*Actually the format is the first three letters of the command followed by the one or two 2-digit line numbers.

**This latter feedback, although not necessary, is useful because the light pen can be put down on a table while the program is executing. The graphical input tablet stylus would remain in place at the current speed.

are properly updated when an arithmetic statement modifies them. The statements PAUSE and END halt execution and the status message at the bottom of the speed line changes from STOP to **STOP**.

The HLL used in this implementation is a subset of FORTRAN IV. Its syntax rules are given in Figure 20. (A knowledge of FORTRAN IV is presumed.) Every statement, except READ and WRITE, takes one execution cycle. In Figure 21 there are some sample statements brightened as they would be during execution. When more than one event might take place, e.g., a logical IF may be false or true, more than one sample is given. A typical display is shown in Figure 22.

Execution errors, such as trying to transfer to a non-existent statement, cause an error message to be typed on the teletype.

3.2.3 Execution Functions. - All the GRAPE execution functions occur in the implementation and the discussion below somewhat follows the discussion of 2.3 on the full GRAPE execution functions. The speed line on the face of the display works like the speed control in a tablet implementation. The teletype is receptive to execution or editing commands only when the speed is at **STOP**. To continue with execution if the speed is **STOP**, one touches some other section of the line.

The \longrightarrow function is performed by typing START followed by the line number; e.g., START 4. As in Section II, one must then move the speed control from STOP to induce continuous execution.

Because there is no easy way of controlling the speed of source program roll with the light pen, two functions have been implemented which move the program forward. Typing ROLL n (e.g., ROLL 12) rolls the program forward n lines, at the rate of two lines per second. The command PAGE displays the next full page, the next 40 lines of program code. As with the full GRAPE design, if execution is begun following a ROLL or PAGE but not a START, the line which was executing when the program stopped will be immediately brought onto the screen.

Typing RESTART puts the first page of the program on the screen; however, it does not undefine all the variables and it does not change the present execution line.

Typing SAVE will do what touching SAVE with the stylus would do. When the computer is not running in the time shared mode, the command SAVE will store on the mass storage device the present version of the HLL program, but control stays in GRAPE.

The above commands correspond to the five functions of 2.3.1. The next few commands correspond to the three functions of 2.3.2. They perform in a manner very similar to the original (Section II) design, and no special examples or discussion are needed here.

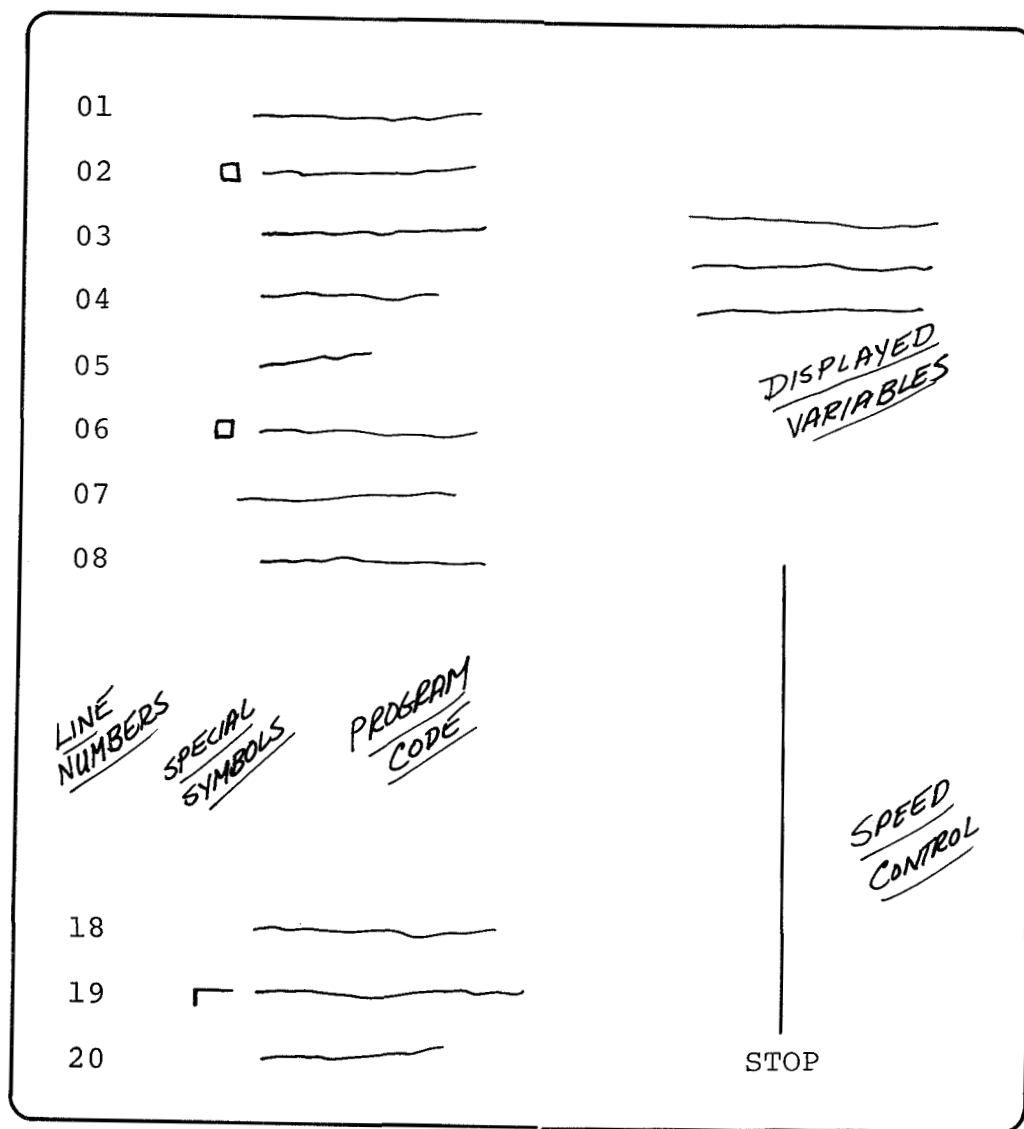


Figure 19. Display Setup

```

C      COMMENT LINES ARE VALID
      A=B
      A=B⊙C
      A=B⊙C⊙d
      A=Z (B,C,D)
      Z (A,B,C)=D
      GO TO SA
      GO TO (SA,SB,SC),A
      IF (A)SA,SB,SC
      IF (A.LR.B)GO TO SA
      CALL SUBRXX(SA)
SA     SUBROUTINE SUBRXX
      RETURN
      READ A
      WRITE A
      PAUSE
      END

```

A,B,C,D are constants or are one-letter integer variables.

Z is an array of dimension 4,20,10.

SA,SB,SC are two-digit statement labels. Any statement
can be labeled in columns 1 and 2.

⊙ is +, -, *.

LR is a logical relation EQ,NE,LT,GT,LE,GE.

Input and output is in I6 format on the teletype.

Only one level of subroutines is permitted.

There can be no imbedded spaces.

Figure 20. Language Syntax

```

C      THIS IS A COMMENT

      GO TO 30

      GO TO (10,20,30),I
      GO TO (10,20,30),I
      GO TO (10,20,30),I

      IF(N)10,20,30
      IF(N)10,20,30
      IF(N)10,20,30

      IF(I.EQ.J)GO TO 40
      IF(I.EQ.J)GO TO 40

      I=J-9*K          =-00012
      Z(3,1,5)=K       = 01488

80     CALL NEWTON(80)
      SUBROUTINE NEWTON
      RETURN


      READ A
      WRITE A

      PAUSE
      END

```

Figure 21. Sample Statements

```
01      I=3
02 C    □ SET ARRAYS
03      IF (N.GT.0) GO TO 10
04      J=K*2+L
05      GO TO 30                X=-00001
06 10    J=K*2-L                I= 00003
07 30    Z(I,J,K)=1
08      IF (L) 40,40,45
```



STOP

Figure 22. Typical Display

One can set a breakline at line n by typing SETBREAK n. This breakline can be crossed out by KILLBREAK n.

The HLL program variables are displayed on the right side of the screen by SHOW VARIABLE n, where n is a number from 1 to 25 corresponding to the variables A through Y. This variable can be removed from the screen by REMOVE VARIABLE n. A maximum of five variables can be displayed at one time. The value of a single-valued variable can be set by typing VARIABLE SET n m where n is the variable and m is the new value.

CONDENSE n m will condense from line n to line m. EXPAND n will serve to cross out the condense character at line n and bring back the hidden lines. Any illegal input, such as the use of a line number which is not presently on the screen, causes an error message to be typed on the teletype.

3.2.4 Editing Functions. - Typing DELETE n while the speed is **STOP** will delete line n from the source program. For example, if one typed DELETE 39 when the bottom of the screen read

```
      .
      .
      .
38      F = 128
39      G = K* 34
40      H = V - W
```

the bottom of the screen would then look like

```
      .
      .
      .
38      F = 128
39      H = V - W
40      I = 6 * I + J
```

Similarly typing INSERT n followed by a FORTRAN statement will push the bottom line off the screen and insert the new statement after line n. One can type CHANGE n followed by the new statement to change a line of code. Finally, typing MOVE n m will move the present line n to just after the present line m. Blocks of code can be moved by condensing them to a single line. Illegal inputs cause error messages to be typed. These four commands correspond to the five functions of 3.4.1. Change serves for both \wedge and overwriting. The compiler features and the two messages described in the last part of 2.4.2 are not implemented.

Setting variables was discussed under execution functions; array entries must be set by a program statement.

some other points are worth mentioning. When GRAPE is first started there is a comment statement on line 1. Otherwise there would be no way to insert the first statement of a new program. If the source program to be run is already on the mass storage device (disk), it will be automatically read when GRAPE first starts. The time-sharing system has a set of special characters which permit the user to erase the last character, the last word, or the last line he has typed. The line numbers in this GRAPE are done differently from the earlier description. Here the line numbers 1 through 40 are always on the screen. They serve as markers for editing and execution but not as reference points in long programs.

3.2.5 User's Manual

SITUATION

MESSAGE

Program Executing

Part of speedline is bright

Program Not Executing

STOP: below speedline, is bright

SPECIAL CHARACTERS

☐ Condensed code

☐ Break line

EXECUTION FUNCTIONS

START N start at
 line N

SETBREAK N break line at line N

ROLL N roll N
 lines

KILL BREAK N erase break at line N

PAGE turn one
 page

SHOW display variable 1 - 25
VARIABLE N A - Y

RESTART beginning
 of program

REMOVE do not display N
VARIABLE N

SAVE save the
 program

VARIABLE set variable N to M
SET N M

CONDENSE N M condense
 N to M

EXPAND N erase ☐ at line N

EDITING FUNCTIONS

DELETE N delete
 line N

CHANGE N S change line N to S

INSERT N S insert S
 after N

MOVE N M put line N after M

S is a program statement

3.3 Implemented Grape -- System Side

3.3.1 Concepts in the Programming. - The GRAPE system is a collection of about 60 Fortran subroutines. Since it was clear from the start that the design of the execution functions and the editing functions would be modified as visual results replaced mental images, a special effort was made to separate cleanly each of the functions into a simple short subroutine. Both comprehensibility and ease of modification were enhanced by this software modularity. (Many of these short subroutines are in fact used as co-routines. That is, after control has filtered down from a driving routine through one or more levels of analysis to the routine which performs the appropriate function, the control goes directly back to the driving routine.)

Another aspect of the modularity in the GRAPE system is that the input/output has been separated from the rest of the system. This was done since changes in peripheral devices are likely in further implementations of GRAPE. For example tablet input and character recognition will be substituted for teletype input. More important is the possible change of display hardware, since the use of the display is pervasive throughout the GRAPE software. Commands and data for the IDI display hardware are handled as variables whose values are set in the initialization section of GRAPE. A change of displays did occur during the implementation, and it was handled with little difficulty, but the two displays were quite similar. Moving GRAPE to a new computer with a differently structured display would be a harder task, but the isolation of the display data should simplify it.

3.3.2 Structure of the System. - Appendix A is a listing of the GRAPE system. In the description below, subroutine names are occasionally included, e.g., EDITOR. The more interested reader may wish to look at some of the routines. A description of the data structure follows the over-all program description and reference to it will make the listing more intelligible. A complete reading of 3.3 should precede one's looking at the program.

The system is divided into four sections: initialization, compilation, execution, function.

INITIALIZATION

These routines set up variables and constants which are used by the other programs. Typical of these are character strings and the display screen size. The routines are set to appropriate initial values of various information in the data structure such as the initial execution speed. They set up variables and constants when bit patterns are relevant for the particular display being used. Finally the initial picture on the display screen is generated.

COMPILATION

The compilation section (READST) takes one statement of a source program and compiles it into an intermediate language which the execution section will understand. It also places the new statement in the data structure with proper reference to the statements which precede and follow it physically (not logically) in the program. Finally, it saves the input character string so that it can be displayed on the screen (PACK). No special compilation is used for condense blocks.

EXECUTION

The execution section is made up of a driving program, XECUTE, which during source program execution does the initial analysis of each statement, including checking for break lines and checking for user induced changes in execution speed. Most of the subroutines in this section handle the execution of a particular statement type, e.g., XCOMNT takes care of COMMENT statements. UNDO has the general function of removing from the screen the intensification resulting from one statement execution and preparing the screen for the next statement. DELAY has the function of delaying while each source program statement is intensified. When GRAPE is running under time sharing, DELAY works by calls to the time-sharing scheduler. When GRAPE is the sole user of the computer, it simply executes a short loop to waste time.

FUNCTION

The function section is structured in a way similar to the execution section. When execution stops, control is transferred to a driving program, EDITOR, which does the initial analysis of any requested function and then calls a separate routine to actually perform the function. For example, DELETE deletes a source statement. These function programs do not update the display screen. They work by changing the data structure representing the source program. REGEN regenerates the screen after the routines have modified the data structure. EDITOR also begins program execution when the speed is changed from STOP.

3.3.3 Data Structure. - Some information about the data blocks in GRAPE will help in understanding how the implementation is actually constructed. Preceding the listing of the program in Appendix A is a listing of the COMMON areas. Since no names are duplicated one may consider these variables and constants to be external to all programs in GRAPE. In IDICOM are constants which are display commands. They set certain features of the display hardware, such as intensity and light pen visibility. In BUTCOM are variables which relate to information which regularly appears on the screen, such as line numbers and the speed control. INPCOM is used by the compilation section as an input buffer and as permanent storage for the character string of each source statement. EXCOM contains character strings which are used during execution, the variables A - Y which are used as data in the HLL

source program (VAR), and temporary information which describes the line being executed. EDCOM contains information used during program execution. This information was set up during the compilation phase and may be modified during the editing phase. SUMCOM has additional information about each source statement. It also has information about statement labels (FORTRAN "statement numbers," e.g., 44 I = 1), location in the display buffer of a statement, line numbers on the screen (e.g., 10 44 I = 1). OTHCOM has more information about line numbers and also has more information about the statement being executed.

Most interesting is the information used during execution. The following data is available for each source program statement.

(a) the intermediate language results of the compilation, namely the type of FORTRAN statement (TYPE in EDCOM) and the program variables in that statement (STATE in SUMCOM)

(b) the original code (STUFF in INPCOM)

(c) the present visibility status of the statement, either visible, preceded by a breakline, deleted, or condensed (VIZ in EDCOM)

(d) the next physical (not logical) statement in the program (POINT in EDCOM)

(e) the present location of that statement in the display buffer (DBUF in SUMCOM) and the present location of that statement on the screen (LBUF in SUMCOM)

Also used during execution is STLABL in SUMCOM which tells which statement has a given statement label. The other large array, LINNUM in OTHCOM, is used in editing. It tells which statement is on a given line of the display screen. Relevant to (d) above notice that there is no backward physical pointer. As a result only forward ROLLing is (easily) implemented. For the same reason, when a new section of text appears on the screen, the presently executing statement is the top line on the screen, not the third or fourth.

3.3.4 Other System Information. - Certain programs called by GRAPE are not part of the GRAPE system. Most of them are part of the time-sharing system and are related to the display.

DSPATT assigns a display to the user.

DSPTRN turns that display on or off.

MOVU2D puts user generated display code into the display buffer (i.e., onto the screen). The new code may be appended to the buffer (to add to the picture) or may replace code which is already in the buffer (to change the picture).

DSPCUT shortens the display buffer, thus removing from the screen the most recently appended information.

DSPSIZ returns the size of the buffer, i.e., the location where the next block of code will be appended.

LPNSBL activates the light pen.

LPNCLR clears the light pen hit buffer.

LPNRD delivers the light pen hit buffer to the user.

RDIN, WRLS, SEARCH are disk input and output.

BN2DEC turns an integer into a character string.

The subroutines in GRAPE are listed below. Figure 23 is a block diagram of the system. Certain very commonly used subroutines such as UNDO and DELAY are not included in the block diagram, which is intended to give a basic feeling for the flow of control in the system.

INITIALIZATION

MAIN
SETCON
SETLET
SETARR
SETBUT

COMPILATION

READST
STNUM
VARIAB
PACK

EXECUTION

XECUTE
PENSEE
XCOMNT
XGOTO
XCGOTO
XPAUSE
XEND
XREAD
XWRITE
XRITH
XARTHF
XARITH
XLOGIF
XYSLGF
XNOLGF
XSUBRU
XSBCAL
XRETRN
XARREV
XVEARR
BRAKE
UNDO
DELAY
EXERR

FUNCTION

EDITOR
STOOP
REGEN
CHANGE
INSERT
DELETE
CONDEN
EXPAND
START
MOVE
SETBRK
KILBRK
VARSET
SHOVAR
PUTVAR
REMPAR
SAVE
OUTLIN
RESTRT
RENTRY
PAGE
ROLL
RANGE
EDERR

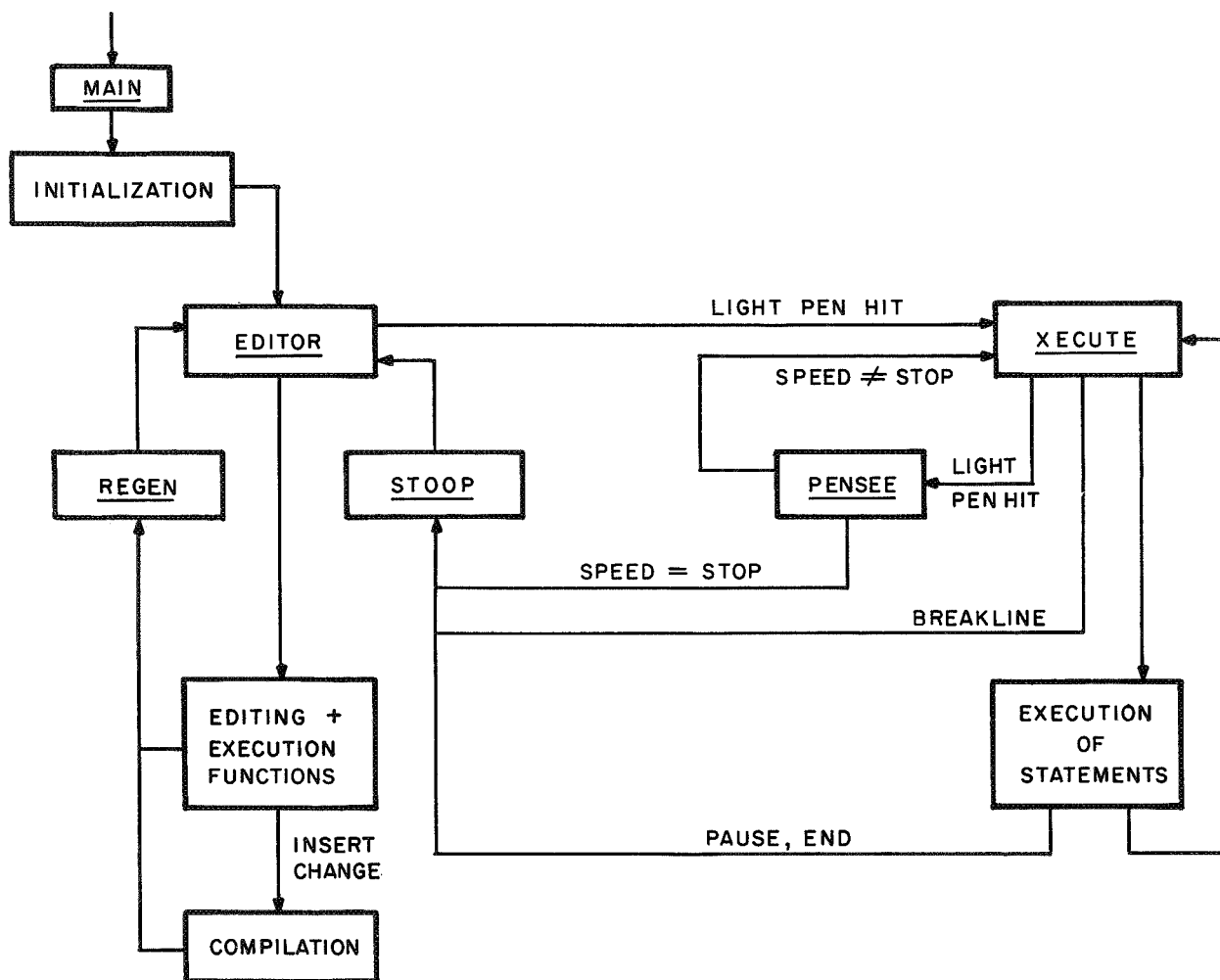


Figure 23. GRAPE Block Diagram

3.3.5 Graphical Input Tablet Interface. - The interface of the graphical input tablet with the rest of the GRAPE system must be described separately since it is not part of the implementation. In the light pen and keyboard implementation during source program execution, the light pen is checked after every execution cycle. Light pen hits which are not on the speed line are disregarded as are all but the most recent one of the hits on the speed line. Similarly, when a graphical tablet is used, after each execution cycle GRAPE will read the tablet input buffer and take the most recent position on the speed line as the current speed. During source program execution GRAPE requests the computer executive system that no ink trace be put on the display, that points be recorded only when the stylus is resting on the tablet, and that the data rate be slowed to about twenty points per second. If the computer system does not understand such requests, GRAPE will have to do this data reduction.

It is easy to tell whether the stylus is on the speed line since the line is at a particular X value with a range of 1/4 inch to either side. The length of the line is divided into about twenty discrete speed areas. If the stylus is resting precisely on a division between areas, the Y value input may alternate between the two areas and the speed of execution will alternate between the two speeds. This could be remedied by requiring that the stylus have moved at least 1/4 inch before a change in speed is recognized. However, the difference between two successive speeds will not be readily noticeable to the observer, and the speed is not likely to remain at any particular setting for very long. Therefore, adjusting for this rare occurrence is not warranted.

When the speed has been set to STOP, GRAPE requests the computer system to record input data points in the manner that the character recognition program prefers. As soon as a single data point away from the speed control is read, all tablet data points go directly to the character recognizer. It returns to the GRAPE system the characters written, including their locations and sizes. The recognizable characters must include all the characters from the higher level language plus \rightarrow , \wedge , $>$, \sqcap , \square , $?$, \hookleftarrow , \uparrow , \sim . The use of graphical tablet data to edit a page of computer program code has already been demonstrated, most notably at Rand (see footnote in Section 2.4.1). GRAPE knows where every character on the screen is and can update source program code, as required.

If no characters are being written, GRAPE does nothing. However, when the last few characters are the name of a variable plus an equal sign at the bottom of the screen, then the value of the variable is displayed. If $>$ is written between two line numbers, GRAPE must adjust the display and set up proper internal pointers for the new line. Likewise when a line is deleted, when a variable is changed at the bottom of the screen, when a breakline is inserted or deleted, when statements are moved or condensed, or when \rightarrow is drawn, GRAPE

updates its internal data structure and the display screen. These are the same actions that were done for these functions in a keyboard implementation. In the case of \rightarrow , if the statement has been edited, recompilation must precede execution. Any character, including a non-recognizable one, that is drawn over a message such as RESTART means that this function is to occur, except that if UNUSED is on, an erase character will turn it off.

Any character drawn at the STOP end of the speed control means perform whatever compiling has become necessary because of changes in code, then execute one cycle. Furthermore, tablet data now comes to GRAPE, as it did earlier during source program execution, instead of going to the character recognizer.

ROLL is a special situation. The user will be holding the stylus down on one of the arrows and GRAPE, not the recognizer, will have to examine the data points. There are two ways in which GRAPE might get control of the input data, depending on the computer system. The tablet input software may have an alarm if the stylus is pressed down for too long, or GRAPE may look at one X,Y coordinate pair every half second to see if the stylus is near ROLL.

There are convenience reasons for using the character recognizer to tell whether the user is pointing to any of the messages except ROLL. First, tablet data is uniformly handled by a single program, thus limiting the amount of programming which must be put in GRAPE. Second, the recognizer is already programmed to adjust for noise from the tablet. Third, the recognizer sees characters only after pen strokes are completed, that is when the stylus is no longer being pushed down on the tablet to trigger the pen point switch. As a result, the request of a function will be known to GRAPE only after the stylus has been pressed on the message, not during the pressure. This solves the problem of accidentally performing a function a second time because GRAPE had completed it the first time before the user had lifted the stylus.

3.3.6 Language. - Some comments are in order for choosing the language for the implementation being discussed. Some claim that using a machine oriented language is the only way to write "systems" while others claim that higher level languages should be used. The latter say that the loss of efficiency and loss of flexibility is more than compensated for by the speed of programming and clarity of programs. Direct transferral of programs from one machine to another is another advantage of HLL, although work such as this is often machine dependent.

The trend is towards higher level languages. For example, MULTICS at MIT is written in PL/I (ref. 11). (I have done systems work in both machine oriented languages and procedure oriented languages and my preference is definitely for the latter.) BCPL, although not perfect, is probably the best available language for the task. Among its useful features are an increased variety of conditional loops, recursivity, freedom in creating data structures, and direct reference of both the contents and the address of variables.

If the computer language revolution involving "compiler-compilers" and "extendible languages" does not occur in the next few years, the inevitable popularity of PL/I will cause it to be the most used language for system work.

FORTRAN is certainly not ideal for systems work. Designed primarily for scientific work, FORTRAN'S most highly developed features, the mathematical routines and the input/output routines, are useless for writing systems. This language has only one kind of loop, has no facilities for letting the user get at the machine hardware, has limited facilities for distinguishing between external and internal variables, and has severe restrictions on the form for subscripts and loop indices. Further, it has no facilities for dealing with interrupts. Both the code produced and the library subroutines are rarely re-entrant.

However, with the help of a few short assembly language routines, plus a few programming techniques, one can write systems in FORTRAN. Typical of the former are routines which permit bit manipulation in the live registers. Typical of the latter is a technique for referencing specific physical addresses in memory.

Given that a HLL language is going to be used for a particular system, and given that for an average size "system" the extra trouble it takes to program in the worst of these languages is less than the extra trouble it takes to write a new compiler, the availability of a particular language may be the determining factor in the decision of which language to use.

The time-sharing system of which the display system software described above is one part is written in FORTRAN. Eighty to ninety percent of the code is in FORTRAN. The reason that there is even that much assembly language code is that the instruction set of the DDP-516 is not large. Thus, saving the live registers of the computer takes about fifteen instructions, while on some computers saving them takes only one.

GRAPE is written entirely in FORTRAN. It consists of many short routines which are essentially self-documenting. The present implementation could be moved to another computer and display much more easily than one could rewrite it from flow-charts, but it is not machine independent. It depends slightly on the structure of the DDP-516, slightly on the display system software which is running on the DDP-516, and somewhat more on the IDI display. Changes to GRAPE to accommodate differences between computers would not be difficult. Changes to accommodate differences in displays would of course depend on how different the displays were.

On the success of writing GRAPE in FORTRAN, it can only be said that the programming was not difficult. (It would of course have been helpful to use GRAPE to debug itself.) Additions and changes to GRAPE which were indicated during on-line experience with the system were easy to make. Finally, it takes relatively little effort for a person unfamiliar with the programs to understand them.

3.4 Results of Implementation

3.4.1 Teaching Tool/Analytic Tool. - The original implementation was a research effort and much that was learned has been reflected in the GRAPE design of Section II.

GRAPE has proved to be useful and effective as a teaching tool because, while a beginning programmer is not at all capable of interpolating backwards and forwards through a program armed only with input/output results, with GRAPE the novice gets individual dynamic instruction on how every statement operates. He gets to try easily his own programs, and GRAPE makes visible all that he could not see. For teaching programming, interpretive execution on a teletype oriented time-shared computer is a vast improvement over batch processing with two day turn around; GRAPE is as vast an improvement over the teletype.

When GRAPE was used, results indicated that no instruction is necessary to teach someone the meaning of the information being temporarily displayed on the screen, as described in Source Program Execution, 3.2. As intended, the user can follow the display without comprehending each line of code. And this is true even though the display is not as clear as it would be in a GRAPE implemented according to the theoretical design. The input to implemented GRAPE is much less natural than it might be (being teletype rather than tablet) but it is not hard to learn. Ten minutes live demonstration and the one page User's Manual are all the information an inexperienced programmer needs. And a half hour experimenting with the system on his own will make him a GRAPE expert.

Although the GRAPE language limitations have kept any large tasks from being implemented with GRAPE, many people have experimented with GRAPE and a few people have used it on real programming problems with favorable reactions. GRAPE seems to be of real value as a program debugging and program modification aid.

The value of GRAPE in two of its three areas of application is thus described. The first area of application is as an aid for teaching programming. We have already discussed how GRAPE helps during the stage when the novice cannot visualize how a program sequence might work, as well as how GRAPE provides the immediate reinforcement so helpful in learning. The second area of application is an aid to understanding and perhaps modifying someone

else's program. GRAPE does everything that one tries to do during a desk simulation of a program's operation. GRAPE does the work faster and it does not make mistakes. Computations and calculations are done by the computer; the user is free to concentrate on where and how the program operates. With the GRAPE execution and editing features, one can modify a program by trial and error, something which one would not do if his only feedback was programmed output.

3.4.2 Debugging Tool/Programming Errors. - The third GRAPE application area is program debugging. A survey was made of several programmers whose programming experience ranged from two months to several years. The programming errors most frequently mentioned as common sources of trouble are listed below:

1. A variable name changed but not everywhere
2. Bad array entry $M(I,J)$ for $M(J,I)$
3. Poorly done input/output
4. Incorrect mode declaration
5. Infinite loops
6. Misuse of subroutines
7. Accidental local use of an externally defined variable, mistake using COMMON
8. Neglected special cases
9. Testing of boundary conditions
10. Wrong relational ($<$ instead of $>$) or wrong logical (doing some operation based on a result of TRUE instead of a result of FALSE)
11. Forgetting simple things, such as initializing or incrementing
12. Wrong order of doing things e.g., computing a result, then resetting the variable, then outputting the variable, instead of doing the output before the reset.
13. Reference to the "wrong variable"
14. Bad arithmetic statement

Indications are that the programming errors in the development of a time-sharing system (not the one GRAPE runs under) and the errors in a lengthy applications program are of these types. All of the errors in the applications program and almost all the ones in the time-sharing system fit into one of the above categories.

The first error is an error which a good editor will handle. It should be possible to say "change all occurrences of character string A to B". When error 2 generates out of range subscripts, an error checking interpretive compiler (as recommended for use with GRAPE) will catch the fault. It is felt by many that error 3 is basically a problem with the language. Better input/output facilities in the FORTRAN language are regularly implemented on individual computers. GRAPE does provide the capability to check "line" oriented input/output such as might be read or written on teletypes or line printers. No help is provided for checking forms of I/O such as binary output to a magnetic tape.

The operation of GRAPE, as described in Section II makes certain errors almost impossible to miss. Errors 4,5,6, and 7 fall into this category. Infinite loops, in particular, are delightful to watch in execution. Figure 24 is an unnecessarily complicated program to do integer multiplication by successive addition. In it is an error in which attempting to square a number causes an infinite loop. The error, and a possible correction, stand out vividly when the program is run under GRAPE as shown in Figure 25.

GRAPE does a good job of calling type 8 errors to the attention of the programmer. Since expressions are evaluated during execution, the particular values which must be separately dealt with are relatively easy for him to notice. Similar to this is type 9 since the boundary conditions are possible special situations. The programmer can watch how these conditions and values are treated by the program and be satisfied that all is well. This is particularly useful if these conditions are not explicitly set by the programmer during the normal (non-GRAPE) execution of the program. Using GRAPE he can preset values which would normally be calculated and can avoid execution of the statements which do calculate these values.

With GRAPE, errors 10, 11, 12 are readily detected because of the feel the programmer has for his own program. If he sees, brightened in front of him, that COEF equals 45.2 and that because of this the routine that handles zero coefficients is entered, he will recognize that he wrote his test statement backwards.* If he sees that the first value of the variable SUM is 25,490, he knows he forgot to initialize it. Likewise he knows where his output is being done, and if he sees his carefully calculated results being set to zero before the output section is reached, the programmer will react.

*This kind of information simply would not be available without GRAPE type operation.

```

C      C=A*B    WITHOUT    MULTIPLICATION

      M=A

      N=B

      C=O

      IF (M.EQ.0)GO TO 30

      IF (N.EQ.0)GO TO 30

10     IF (M.LE.N)GO TO 20

      C=M+C

      N=N-1

      IF (N.GT.0)GO TO 10

      GO TO 30

20     IF (M.GE.N)GO TO 10

      C=N+C

      M=M-1

      IF (M.GT.0)GO TO 20

30     PAUSE

```

Figure 24. Program With an Infinite Loop

Cycles 1,3,5,... 10 ☐ IF (M.LE.N) **GO TO** 20
 20 ☐ IF (M.GE.N) **GO TO** 10

Cycles 2,4,6,... 10 ☐ IF (M.LE.N) **GO TO** 20
 20 ☐ IF (M.GE.N) **GO TO** 10

Figure 25. Execution Of An Infinite Loop

The GRAPE display technique helps the user detect type 13 errors by calling his attention to the line as it is being executed and by displaying the value of each variable. If one sees his own program say IF (2 .EQ. 25340) when something approaching equality was expected, he will suspect something is amiss. Errors of type 14 usually result in values which are thousands of percent off; wrong variables or a neglected division are the typical mistakes.* GRAPE makes these errors impossible for the programmer to miss. For the rarer type of arithmetic mistakes, such as initializing pi to 3.17, only hand calculations, with GRAPE doing the computer work perfectly, will catch the error.

*Personal experience plus programmer survey.

IV. TIME SHARING HIGH SPEED GRAPHICAL INPUT DEVICES

4.1 Central Problems

Several major design problems must be met when time sharing high-speed graphical input devices. Identifying these problems and the discussion of the effects of the feasible solutions outline the requirements for the graphical input section of a time-sharing computer executive system.

There are some problem areas in the time sharing of high-speed graphical input devices which are independent of time sharing. First is the fact that input is occurring. Input and output are more complicated than straight computing because the characteristics of the peripheral device must be considered as well as the characteristics of the computer. Second, the input is being produced directly by a human and so the relevant characteristics of human beings, for example the jitter when someone draws a "straight" line, must be considered. Third is the amount of computation required. Whatever computation is necessary to accept and store input data must be repeated hundreds of times a second. Even a few microseconds of computing for each data point will consume a large percent of the central processor's computing time. One hundred microseconds of computing repeated 1000 times a second would be 1/10 the computing power of the machine. Fourth, graphics software takes a lot of computer space. A large buffer is necessary to store data which may come to the computer at the rate of 1000 points per second. If this data is to be instantly available for analysis it must be readily accessible and must stay in physical memory rather than be put on a storage device. The programs to analyze data from a high-speed input device are also long and complex, and if one wishes to analyze the data at such a rate as to give the human user real-time response to his input, then these programs must also be rapidly accessible.

There are problems which arise because of the time-sharing environment which must be solved satisfactorily for the graphical section of the executive system as well as for any other section. One must ensure that one user cannot affect the data or the equipment of another user without explicit permission. One must ensure that scheduling for service of the various graphical devices is done fairly so that one user does not inadvertently get better response than another, although all graphics users require better response from the system than the teletype users require. One must decide how (or if) each user will be assessed for the services provided.

Some problems occur because both time sharing and graphics are involved. These all have to do with the real-time requirements of the graphics hardware superimposed on a time-sharing environment which cannot guarantee access to the central processor to a particular user at a particular time. With input occurring at such a high rate, the time it takes the system to switch users makes it impossible to give control to the graphical user to let him process each data point as it arrives. Therefore, the executive system must assume responsibility for controlling the device and maintaining an input storage buffer that contains the input data from the device and can be read by the user.

In general purpose time-sharing systems, the executive system rarely performs a function for the user which the user could do himself. One reason for this is that making programming changes to the executive is dangerous when many users are depending on an error free system. Another reason is that time during which control is in the executive is often considered overhead, and the cost for this time is shared by all users. Finally, in a time-sharing system when input/output, including paging, is required by one user some other user can operate, but when it is required by the executive itself no user will run. But there are functions concerned with graphical input devices which could be performed by the user that are better performed by the executive. The following three sections deal with functions related to graphical input devices which the executive of a time sharing system must do, functions which the executive ought to do in order to provide good services for the users of graphical devices, and functions which the executive system might do even though the users could also do them.

4.2 What Must Be Provided

As with any peripheral device, a particular input unit must be associated with a particular individual; this allows the individual to refer generically to the device type from within a program rather than forcing him to specify the physical number and it prevents one individual from (accidentally) reading or destroying data which was meant for someone else.

As with all input/output, the system executive will control the device. This involves responding to the interrupts and accepting the input data. It also involves sending control signals to the device, often at the request of the user. These would include turning the device on or off, perhaps changing the rate of interrupt, perhaps setting some mode control within the device.

As with other peripherals, the executive system responsible for the graphical input must keep track of usage so that each user can be properly charged for usage by the system.

Another function which the system must perform comes about because these devices may be interrupting the computer as often as every millisecond. The system must collect the input data and store it away until the user gets a chance to process it. The simplest thing for the system to do is to retain some fixed amount of buffer space and to add data words into this space as they arrive from the device. When the space is full, the executive will collect no more data. The user is responsible for emptying this buffer through specific system requests and for informing the system that it may once again collect data.

Under certain circumstances this scheme works perfectly well. The advantage is that it is fast and requires as little system code as one could have and still operate in a time-shared fashion. The disadvantage is that it uses a lot of space. For example, although the user may get access to the computer on the average of every three seconds, he may occasionally be kept off for twice that long. If each data point is three computer words long, for the two coordinate axes plus a word of control information, and the input device generates data once per millisecond, then 18,000 words of data will have been sent to the computer in the six seconds between his two turns at the machine. Further, even if that much data could be saved by the system, if the user could process only five seconds of input data during his time slice and could not declare the buffer empty, then no data would be collected by the system while he was waiting for his next turn.

For many uses of graphical input devices, the user will produce only two or three seconds worth of data and will then wait for a response from the computer. Also, one could equip the computer with a lot of memory, and when a graphic input device was being used, devote much of it solely to storing graphical input data. These are not unreasonable solutions, for even complicated curves can be drawn in two or three seconds, and a time-shared computer always can use extra memory.

But the problem is not always so easily solved. In the following paragraphs other ways the executive might handle the input storage buffer will be considered.

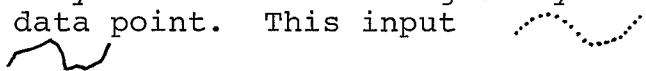
4.3 What Ought To Be Provided

4.3.1 Feedback. - The first function the executive system should provide to the user is feedback. If the user is writing characters, if he is drawing pictures, or if he is manipulating an image on the display, he must be able to see what he is drawing before his program has had a chance to analyze his input. There are certain times when one is interested solely in inputting two dimensional information to the computer. For these uses, the graphical tablet can be used without a display. The Sylvania version of the tablet allows one to put a piece of paper on the tablet and use a real pen to draw. However, most uses of graphical input involve graphical output via a display, and it is on the display that the feedback occurs.

The user program can interpret and deal with input data in any way it sees fit. But for the few seconds between user turns on the machine during which the system must provide the feedback, a linear mapping from the surface of the tablet to the area of the display reserved for tablet input is sufficient to show the user what he is drawing. In the simplest case, the tablet surface is mapped to the entire display face. This lets the user

"write" anywhere on the display. For the wand there is the problem of representing three dimensions on a two dimensional surface in such a way that the user can continue to draw.

Whatever the display mechanism, information must be put out on the display that corresponds to the information presently in the input storage buffer. This temporary information is deleted from the screen at the same time that the user takes the data from the input storage buffer. Since the coding necessary to put points on a display is different from the coding of the points which the tablet gives to the computer, a separate feedback buffer is needed. This feedback buffer has points recorded in display format and is put on the screen every time the user's display is refreshed.

The information in this feedback buffer is used only by the system, never by the graphics users, and so the form of this information can be adjusted to reflect characteristics of the display being used. Such action is desirable because for a given display some codings of this point information may take even more space than the input storage buffer takes. Also, for a given display some codings of this information may take longer than a single refresh cycle to put on the screen. For example, if the display has a constant time vector generator (one which takes the same amount of time to draw any length vector), one might display the stylus trace by vectors connecting every fifth data point rather than every data point. This input  might temporarily appear as

This scheme would save both display time and feedback buffer space at a rather small cost in system compute time. If the input buffering technique being used is to keep loading the buffer until either it is full or the user is ready to process data, then this scheme works well. The feedback buffer must contain information for exactly one fifth the number of points that the input storage buffer can contain. It will be seen later that it may be desirable to have a much more complicated handling of graphical input data.

4.3.2 Data Compression. - The technique described above can be employed because of an important feature of graphical input which distinguishes it from other high-speed input such as magnetic tapes. There are only small differences between successive data points compared to the possible difference between two random data points. This feature of graphical input can also be used to save space in the input storage buffer. For each data point, one must store only the difference between it and the previous data point. With the high sampling rates which these devices need in order to be effective drawing tools, it may take only one half a computer word to describe the difference in all the spatial coordinates plus the device mode (e.g., the position of the pen point switch).

Thus, half a word might easily serve to hold information previously requiring three words: X coordinate, Y coordinate, mode.

Although optimizing space might not seem crucial in temporary buffers, these are buffers which may run to 10,000 or more words, and which, by their real-time requirements, must probably remain in physical memory. Therefore, space savings by a factor of 6, 4, or even 2 is worth considering. It does however take computer time to effect this space savings, but one other fact settles the situation in favor of using this simple form of data compression. For drawing, the graphical input device user is himself primarily interested in relative differences, and he would have to compute the differences if the executive system did not.

There is one other function that the executive system can perform on all the data points which early use of the graphical input tablet has demonstrated is valuable (ref. 12 page 8). This is averaging each data point with the previous several data points before recording it. This has the double effect of reducing the slight jitter one has while drawing and of minimizing the importance of hooks at the beginning and end of lines.

4.4 What Might Be Provided

4.4.1 General. - There are three areas in which the executive system might reasonably perform functions which could be performed by the user when he has control of the central processor. Whether or not these are actually done by the executive depends on two things. First, the amount of time the systems programmers are willing to spend making the graphical devices convenient to use. Second, the extent to which the executive system is willing to give extra time and access benefits to the graphical device user under time sharing. A poorly supported graphical input device will be practically unusable in a time-sharing system. As long as these devices promise to aid in problem solving, it is false economy to oblige users to work from the less demanding teletypes. And lack of software support is an effective way of denying access to these devices.

The types of functions which the system offers to perform for the user are functions which it appears all graphical input device users will want and would have to do for themselves. It should be true that there is some over-all savings (in space or time) by having the executive system perform the operation on its time rather than have the user do it. Thus, data averaging is something which will be done by all users. Furthermore, it would cost each user a pass through the data to do the averaging, but it can be done by the system while the data point is being stored away.

The three areas are (1) more complicated data reduction, (2) more complicated buffering to permit continuous input, (3) mode analysis.

4.4.2 Data Reduction. - The data reduction which the executive system might perform on the data points from a graphical input device may involve reducing the precision of the data and/or reducing the quantity of the data.

If the data from the device has more bits of accuracy than the user needs, he can perhaps save space in the computer and save time during his own data analysis by requesting that the precision of the data be reduced. This is equivalent to dividing the range of data into discrete areas. Reduction of precision can be done on each data point as the point is received by the computer. Throughout this section the four data points

(18,15) (27,36) (29,70) (15,68)

corresponding to four successive X,Y coordinate pairs from a graphical input tablet will be used as an example. If the user requests that his input data be reduced in precision by a factor of 10 in the X dimension and 5 in the Y dimension then the resulting data points are (10,15) (20,35) (20,70) (10,65).

Here one must realize that there is another dimension beside X and Y which is being recorded implicitly. This is the dimension of time. In the analysis of data from graphical input devices time can be as important a dimension as any of the spatial dimensions. It need not always be true that spatial coordinates are measured and recorded, while time serves as the metered dimension whose value is known implicitly. For example, one records the times when a racer completes each lap of a race, and one can conceive of computer controlled devices which are metered along some other dimension than time.

A reduction in precision along the metered dimension simply involves recording only every Nth data point. For many purposes, such as drawing straight lines or pointing instead of drawing, the data rate of the graphical tablet is higher than necessary, and a reduction in the precision along the time dimension will certainly save computer space and computer time for both the executive system and the user. In the example of this section a reduction by a factor of two in the precision on the metered dimension results in the data points (18,15) (29,70).

The precision could be reduced on both the measured dimensions and the metered dimension giving data points (10,15) (20,70). Furthermore, reducing the precision of the data may be done whenever the quantity of the data is being reduced.

Reduction in the quantity of data is more complicated, and it will be the topic of the rest of this section. This reduction involves selectively discarding data according to user supplied specifications. The reduction must be applicable at the time the data point enters the computer, and it must not require any searching back through previous data. What we are looking for is an indication that some data is redundant. But we must know this before doing any real analysis of the data. The single general criterion which leads to such reduction is whether the present data point is sufficiently close to the previous recorded data point to be partially or totally disregarded.

The actual numbers which determine whether one data point is close to another must be supplied by the user, but there are various ways in which this type of reduction can be applied. It may be that a data point is close enough to the last recorded data point to be disregarded if the difference between their values in a given single one of the measured dimensions is smaller than a user supplied number, or the two points may have to be close along a combination of the dimensions. For each of these two situations the data point can either be disregarded completely or its value along the other dimensions can be recorded. These four situations will be treated in the following paragraphs. When data points are occasionally being discarded completely, care must be taken not to lose the value of the non-discarded points along the metered and implicitly recorded dimension. Specifically, this means that a count must be maintained of the number of data points in a row that have been discarded completely. This count is put into the input storage buffer preceding the next data point that is recorded there.

The first situation is using a single measured dimension to invoke the data reduction and recording all other dimensions. Here each data point is compared with the previous recorded data point along the chosen dimension, say X coordinate to X coordinate. If the difference is less than some value previously supplied by the user, then only the values along the other dimensions are recorded. In the example of this section if 10 in the X dimension were the criterion then the data recorded would be (18,15) (,36) (29,70) (15,68). That is, the X value of point two was sufficiently close to the X value of point one to warrant not recording it. Had 15 along the X dimension been the criterion, the data recorded would have been (18,15) (,36) (,70) (,68). In the input storage buffer, a reduced data point should take less space to record. Of course some mark must be used in the buffer to indicate that the present point is a reduced one. As mentioned above, this form of data reduction could be applied to data whose precision had already been reduced.

The second situation holds if it is decided to totally disregard points which are sufficiently close. In this case, the system will put a count of discarded points between recorded points. For the example in the middle of the previous paragraph the recorded data would be (18,15) 1 (29,70) (15,68). For the example at the end of that paragraph the recorded data would be (18,15) 3. It is anticipated that these counts will often be much higher. This is apparent if one considers that holding a graphical input stylus stationary for just a few seconds will generate thousands of data points all of which are almost identical. Once again, this reduction can be applied to points of reduced precision.

For situations three and four, closeness along some combination of measured dimensions is what determines redundancy. The question which must be answered is what do we mean by combination. It may be closeness along at least one of the chosen dimensions; it may be closeness along all of the chosen dimensions; it may be closeness according to some function of the chosen dimensions.

Both of the first two can easily be provided by the system. Specification by the user is simply a critical value for each dimension being tested and an indication of whether the logical results of the tests are to be OR'ed or AND'ed. If in our example the critical X value is 5, the critical Y value is 5, and the results are to be OR'ed, the recorded data is (18,15) (27,36) (,) (15,68).^{*} Had there been a Z dimension, its value would have been recorded in all four data points. Also had the fourth data point been (15,33), then the recorded data would have been (18,15) (27,36) (,) (,) with separate recordings for both the third and fourth points. If the results of the closeness tests were to have been AND'ed together, then none of the data points would have been partially disregarded. If instead of only partially discarding redundant data points as has been done so far in this paragraph, we totally discard them, then the first OR example is recorded as (18,15) (27,36) 1 (15,68); the second OR example is (18,15) (27,36) 2; the AND example still has no points discarded. This last reduction, with equal critical values on all measured dimensions, the results AND'ed together, and redundant points totally discarded, has a simple geometric representation. Imagine a square around the first data point received from a graphical input tablet. As long as the tip of the stylus remains within that square the data points are simply counted. When the stylus moves out of the square, a new X,Y coordinate pair is recorded and a new square is imagined.

^{*}Note that point four is sufficiently close to point three but point three is not recorded on the chosen dimensions.

The third possibility mentioned above about a combination of dimensions is closeness along some function of the dimensions. Implementation within the executive of such data reduction is reasonable if many users will be taking advantage of it. The percent of users which constitutes "many" depends somewhat on the difficulty of the implementation and hence the burden that the executive system and, indirectly, all the users have to support for the existence of any particular executive function. With present devices and applications, no such data reduction function of multiple measured dimensions belongs in the executive of a general purpose time-sharing system. An example of a function which is easy to implement but few people would use is one which lets AND and OR and parentheses be used in the specification. Closeness could then be defined as some expression such as

$$(\Delta X \leq 10 \text{ OR } \Delta Y \leq 15) \text{ AND } \Delta Z \leq 15$$

An example of a function which more people would use but which is much harder (in the overhead sense) to implement, is that two two-dimensional points (X_1, Y_1) and (X_2, Y_2) are close whenever

$$\sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2}$$

is less than some user supplied value. This computation defines closeness as a circle around (X, Y) similar to the square used earlier. Only if almost all the graphical input device users of the system were going to use this technique would it be put into the executive.

There is a degenerate case of data reduction. It is the total elimination of one of the dimensions. On a measured dimension it can be done by reducing the precision of that dimension to zero. However, if such reduction is going to happen often, either because one drawing device is going to be used in lieu of another drawing device of fewer dimensions (e.g., a wand to simulate a tablet) or because some general purpose device which functions as a multi-dimensional graphical device is often run at less than maximum capability, then the supervisor could easily eliminate that coordinate during input and shorten the vector representing each input point by one unit.

One can also disregard the value of the metered dimension but still maintain the sequential order of the data points by not counting points which are totally discarded due to the other forms of data reduction. The effect of this scheme with the drawing devices is to record where the stylus was but not how long it was there. This method of recording data may be quite sufficient for some graphical input device applications.

4.4.3 Improved Buffering. - There are three ways in which more sophisticated buffer control can be usefully employed. The first is a simple change to the feedback buffer. The second is a radical change to the input storage buffer and then by necessity to the feedback buffer. The third is a change in the method of reading the input storage buffer.

First is the feedback buffer. This change is applying the data reduction principles of the previous section to that buffer. The buffer provides vital feedback to the user while he is drawing but before the input stream has reached his program for analysis. We have already noted that there is no need to maintain the accuracy in the feedback buffer that the system maintains in the input storage buffer. In fact, the system designers can choose just when two successive data points are sufficiently close to one another that displaying the second of them for the few seconds will not be of any benefit to the user. This method of data reduction in the feedback buffer shows what is being drawn as an evenly spaced dotted curve. This is preferable to displaying every N^{th} point, which would produce heavy patches of light at corners where one draws slowly, and only occasional dots in the middle of lines where one draws rapidly.*

The decision as to whether to spend system time doing this reduction for the feedback buffer depends solely on the characteristics of the display hardware. If the display has a relative dot function such that information which is going into the feedback buffer takes little space to record and little time to display, then data reduction is not warranted. But if the feedback buffer takes a lot of memory, or if displaying the buffer taxes the capabilities of the display, causing flicker in the rest of the picture, then data reduction is worthwhile.

The second change to the buffers is more complicated. During normal operation device input information is appended to the bottom of the input storage buffer at regular intervals by the executive

*One counter example is a physiological test in which one is required to hold the device as steady as possible and in which the brightness (due to multiple points) on the screen provides the feedback. Precise instantaneous feedback is necessary. The user should be able to turn off all special functions provided by the executive system as long as the system can somehow limit the overhead in time and space incurred by any one user. One does not, however, have difficulty in finding an example so demanding of computer resources that time sharing is contra-indicated.

and information is depleted in large chunks from the top of the buffer at irregular intervals by the user. If the user gets to run the machine on the average of every three seconds he must be able to process at least three seconds of input while he is active. With this method of buffer control, input must stop when the buffer is full and not begin again until the buffer has been completely emptied.

But it would be convenient, and for some applications it might be vital, if input could be continuous even allowing for the vagaries of time sharing. The method to use is a wrap-around input storage buffer with separate pointers to indicate the next location to be written into by the executive and the next location to be read from by the user. The word in an input storage buffer of length N to be read by the user or written by the system after word N is word 1, although when the user asks the system for data from the buffer he gets it in its proper order. The system will not record data in the buffer if the write pointer catches up to the read pointer, and it will not transmit data from the buffer if the read pointer catches up to the write pointer. As long as the buffer is able to hold, say, twice as much input as normally arrives between two successive user opportunities at the central processor, input will be continuous and can proceed simultaneously with analysis. The user can take data from the buffer at whatever rate he chooses. This is easier for the user and will improve his interactive response as well.

A wrap-around input storage buffer can only function with a wrap-around feedback buffer. And unless the nature of the data in both buffers matches in format, deleting code from the feedback buffer takes additional information in the input buffer. There must be pointers from the latter buffer into the former to indicate how much code can be deleted.

Finally, there is a change that can be made in the way the input storage buffer is read. Because the storage of raw data takes a lot of computer space and because the user may deduce that while he can analyze some of the data which he has read from the buffer he cannot yet analyze the rest of it, the user may find it convenient to separate the two functions of reading information from the buffer and deleting information from the buffer. For example, a user might read 200 words from the buffer, discover that he could process the first 150 words but did not yet have enough input data to process the last 50, and delete only 150 words. Then the next time the user read information from the buffer, the first 50 words he read would be the ones which he did not delete.

4.4.4 Mode Analysis. - The third major area in which the time-sharing system executive might assist the graphical input

device user is called mode analysis. Beside the coordinate information available from graphical input devices, most such devices have just a little more information which will usually be interpreted as control information by the user. Most standard light pens have a spring loaded two position switch. The common use of this switch is that the pen reacts, in either a software or hardware sense, to light only if the switch is pressed. Both the Sylvania and the Bolt, Beranek, and Newman commercially available versions of the graphical input tablet have a pair of two position markers which are read into the computer along with the X and Y information and provide some slight indication of the distance of the stylus from the surface of the tablet. One simple use of this Z information is to disregard data if the stylus is not directly on the tablet.

These few extra bits of information are often used to indicate that certain data is to be disregarded and the executive system can work to both its and the user's advantages by letting these bits determine the general mode in which the device will operate. No such information is derivable by the system from simple analysis of the basic coordinate data, thus only these mode bits are singled out for special treatment. The user is permitted to specify what he would like the system to do for various configurations of these bits.

Broadly looking at the role of the executive system with regard to graphical input devices, it accepts data from the device at a certain rate, does some data reduction on the data, records it in an input storage buffer, records it in a feedback buffer, and transmits it to the user.

First of all, the data input rate can be different for different settings of the one, two, or three bits which constitute the mode value. Typically, one might not be very interested in input values unless the stylus is against the tablet. Therefore, one might reduce the input rate to ten points a second until the mode value indicated that the stylus was once again down. Even though two bits of mode information means there are four different mode values, a user could be interested in just one of the bits. He might also set one input rate for mode value equal 0, a second rate for mode value equal either 1 or 3, and a third rate for mode value equal 2.

Similarly, one might wish to change data reduction techniques when the mode changed. The test for mode is made as each point enters the computer and the change in data reduction can be made at that time. Since the mode value is recorded in the input storage buffer along with the coordinate values, the user will know when the change occurred.

Third, perhaps information should be recorded in the input storage buffer only for certain values of the mode bits. As usual, a count is maintained and each time a point is recorded in the input buffer, the total number of non-recorded points since the last recorded one is stored with it. With two reasons for disregarding points, closeness and mode value, the system may keep and record two counts instead of one.

It is even easier to apply mode value analysis to the fourth system function, recording in the feedback buffer, since there is no necessity for counting unrecorded points. A simple test for appropriate mode value is made before the point is put into the buffer.

The mode value can also affect the fifth system function by determining when the input storage buffer is transmitted to the user. The program which uses a graphical input device will often be in a situation of input wait, that is, waiting for a peripheral device to complete input before processing can continue. The graphical device input wait is different from input wait for, say, exactly 100 words from a magnetic tape, because in general only the user can tell whether he has enough input data from the graphical device to analyze that data. However, if the mode value provides the clue that sufficient data is in (e.g., data will be processed as soon as the stylus is lifted from the tablet), then the user can put himself into input wait and the executive system can "awaken" him as soon as the proper mode value appears in the input stream (or as soon as either buffer is full). Thus the user will be scheduled to run only when he is sure he has some computing he can do.

V. EXPANSION AND CONTRACTION OF GRAPHICAL PROGRAM ANALYSIS

5.1 More Powerful Hardware and Software

GRAPE is a display oriented system for higher level language program analysis. It is worth trying to anticipate developments over the next few years in computer displays and in computer languages to see what effect they might have on the GRAPE method of program analysis. We shall first investigate whether improvements to the hardware could or should affect GRAPE. These improvements are ones which change the user's view of displays, not simply improvements to the display circuitry.

The enhancement to displays in which research is currently being done is the addition of the third spatial dimension (Van Dam's work at Brown University and refs. 13, 14).^{*} While this work is of interest in many graphics areas, it does not appear to be useful in program analysis. An exception to this might be in looking at three-dimensional data structures. Since "real" objects are all three-dimensional, such capability is valuable in display work. However computer program data structures can be any number of dimensions, therefore this use of three-dimensional displays is limited. A discussion of the failure of an attempt to use three-dimensional displays to view four-dimensional objects can be found in reference 15. The ability to have some information coded in the third spatial dimension is valuable, but this can be done using display features such as variable intensity, winking, or multiple colors, and does not demand a true third dimension which is as complex as the two spatial dimensions of the display face.

A computer program is one dimensional in that it consists of instructions to a computer to be executed serially. The standard way of writing programs takes a second dimension, with statements written horizontally and aligned vertically. Programs are always written in two dimensions, such as on a piece of paper. The languages themselves are designed to be written in two dimensions. The third spatial dimension is not used in writing computer programs.

A program structure such as

```
IF  X  <   0  do  A
IF  X  =   0  do  B
IF  X  >   0  do  C
```

does not add a third dimension of time because no true simultaneity of operation is implied. The dimension of time is used in writing computer programs only to determine the ordering of instructions in the first spatial dimension.

One could consider using the third spatial dimension to represent the time dimension while the program is operating. Rather than modifying a single two dimensional plane, every

*

The ability to draw patches of light, rather than just beams, is also being investigated, but this is not of value to a display system concerned primarily with characters.

execution cycle could add a planar picture in front of the previous planes. But people do not work this way. Depth information cannot be sorted out the way lateral information can. Individuals do not choose to look at objects which are set up on the visual Z axis if they could instead be set up in a single plane at a constant distance from the eye. Both the obstruction of far objects by near objects and the required muscular change of eye focal depth are causes of this. The conclusion is that three dimensional displays do not offer an improvement in the techniques of computer program analysis.

We next investigate improvements in the languages used for programming computers. Again improvements refer to the user's view of these languages, not the methods by which they are compiled. Flowchart programming, that is specifying a program by a flowchart rather than by a serial list of written instructions, has been discussed and partially implemented in the past few years (Project Grail at Rand Corporation and refs. 16, 17). The general use of flowcharts by programmers could indicate that flowcharts are a more natural way to write programs than are lists of instructions. Flowchart programming is similar to other higher level languages in that from the user's side it is computer independent, and the precise method of implementation of each flowchart instruction is not of direct interest to someone doing program analysis. The differences between flowcharts and the languages we have been discussing are that some of the syntax of the flowchart language, especially control functions, is in the form of shapes rather than characters, and that the language is two dimensional in representation. Within nodes of a flowchart, one will continue to write expressions and other instructions which have no two dimensional analogue.

This improvement in programming notation does not alter the type of information a flowchart programmer would want for program analysis, and many of the ideas of GRAPE would be directly usable. Visually presented timed execution of the source flowchart and extensive control over the rate of execution are still important concepts. So is the idea of condensing a collection of nodes to increase speed of execution and improve visibility. One will certainly be interested in inserting break points, saving the present flowchart, and restarting the execution of the flowchart.

The displaying of the operating flowchart is seriously complicated by the fact that the program extends in two dimensions. One simple solution is to permit rolling in both dimensions much as one might scan a map. However, distances need not be preserved on a flowchart (or on any "graph") the way they must be on a map, and moving nodes around so that the maximum amount of information is on the screen at any one time should be done by the system. Since the density of code per square inch is not normally as high in a flowchart as it is in a linear program, zooming in and out

towards the program by changing the size of characters and the closeness of nodes would be possible. In fact this will probably be necessary since condensing a set of nodes will be natural to do only if all the nodes to be condensed are on the display face at one time. When the program jumps to a node of the flowchart which is not on the screen, a decision will have to be made as to what nodes of the flowchart should be displayed. Most programmers are in the habit of drawing flowcharts from top to bottom and left to right; therefore, putting the current node near the top left is a good first guess. But clearly the display techniques will not be easy to develop.

Since flowcharts are two dimensional, flowchart programming will develop only where two dimensional input is available. The problems of editing will probably already have been solved on such a computer system. The primary difficulty in performing GRAPE type analysis on flowchart programs will undoubtedly be the display handling discussed above, but flowchart languages would seem to have enough in common with other higher level languages to make GRAPE useful in flowchart program analysis.

5.2 Less Powerful Hardware

5.2.1 Display, Light Pen, and Keyboard without Graphical Tablet. - Reductions in the capability of the input/output equipment and their effect on GRAPE will now be examined. The hardware and software blocks necessary to run GRAPE as described in Section II were outlined in 3.1. Certain of these features can be eliminated, and a restructuring of the others defines a system which still retains sufficient power to serve as the tool for program analysis.

GRAPE shows exactly what the source program is doing as it executes, as opposed, for example, to a scheme which only shows which statement is executing. Therefore, it must understand each step of the source program and must present this information to the user; so both the properly constructed compiler and the execution phase of the system are necessary.

GRAPE is display oriented. Even though the material on the display face is characters arranged in lines, no line printer could produce the information, including context, which GRAPE requires. The noise, the speed, and the movement of the paper make any line printer unfeasible as a substitute. Certain hard copy generators produce sheets of paper that contain all the information which would be on a display face. Hundreds of sheets of paper, each of which shows a new execution cycle, could substitute for a "soft" display. But this approach overlooks the fact that these devices actually are displays with additional equipment added to produce the hard copy. Also, the fastest hard

copy generators take ten to thirty seconds to produce a picture. Therefore, the display is necessary to GRAPE, and so is the display support software.





However, the other major piece of hardware is not as vital for GRAPE operations. The graphical input tablet is not yet a common device. Furthermore, the software support, including character recognition, for this device in any particular computer installation is time consuming both for the system programmers to produce and for the run-time computer system to use. One should investigate whether another input device can reasonably be substituted for the tablet. The input device regularly associated with displays is the light pen. It is much less demanding of central processor time to operate and is much more frequently available than the graphical tablet.

The primary difference between the two input devices is that the tablet is a drawing device and the pen is a pointing device. Although the drawing of straight lines can be done with a light pen and tracking cross, the light pen cannot be used to draw small characters. However, it will have been seen that use of the graphical tablet in source program execution, execution functions, and some of the editing functions is strictly as a pointing device. For changing characters in the source program code a keyboard, in conjunction with a light pen, can be used.

Using the light pen as the major input device keeps the user's attention and activity on the display face. Thus he is still interacting directly with his program and not with an intermediate system. Source program execution occurs as described in 2.2. The same factors which led to the earlier design of the speed control apply here. A speed line like the one used in the implementation of Section III (Figure 19) is what is needed. If possible, a mechanical structure should hold the pen in position while the program is executing. Pens are normally attached to the right side of displays and a speed control on the left would interfere with the line numbers. As a result, this control must be on the right. The speed control works just as it did in Section II; moving the control to STOP or to and away from STOP has the same effect that was described earlier.

On most light pens is a push button. Pressing that button while the pen is at either end of the speed line causes a function which was described earlier. At the fast end, all delay between statements is eliminated. At the STOP end, a single cycle of execution occurs. If there is no such button, the top of the speed line must cause execution at the high speed. The slowest continuous execution speed is slow enough that one could touch it and then touch STOP to perform single cycle execution.

Functions are represented by messages on the screen as in Figure 26. Touching a message with the light pen causes the function to occur, just as it did with the graphical stylus. Once again a single device is used for speed control and for function control. For many of these messages, one or two line numbers must also be touched. If a new function is touched before all the line numbers for an old function have been input then the old function is not performed. Feedback for these functions which need line numbers is provided by brightening the function until the line numbers have been touched.

When the source program is executing, none of the light pen messages are displayed. When the speed is set to STOP, the top of the screen looks like Figure 26. RESTART, SAVE, and TRASH are the same as described in Section II. ROLL is as in Section II, except since the light pen is not fine enough to allow multiple rolling speeds, ROLL is a constant two lines per second. PAGE, as described in 3.2, allows rapid movement through a program. Touching →, , or  followed by one, one, or two line numbers respectively institutes the function. If the line which is touched following  or  already has that function, then the function is removed from the line.

The last three messages involve the keyboard. With these either the carriage return or the line feed is used to terminate keyboard input. If one touches VARIABLES and then touches a presently displayed variable, it will be removed from the screen. If one touches VARIABLES and then types the name of a source program variable, it will be displayed. If after typing the name of a variable one also types a value, the variable is set to the value. After touching REFERENCES one can type the name of a variable to see all references to that variable, a statement label to see all references to that statement, or a carriage return to see the unused areas of the program. Touching EDIT followed by a line number allows one to edit that line from the keyboard. The intraline editing instructions from QED (ref. 18), the SDS 940 time sharing system editor, are a good set of editing instructions. QED is based on TECO, which is a display editor designed at the Massachusetts Institute of Technology. Its instructions are appropriate for fast editing where feedback from a display is available. Touching EDIT followed by two line numbers moves the first line after the second line. However, when the two are successive lines, then space is opened to permit inserting a line from the keyboard.

GRAPE, in this configuration, preserves the notions of complete feedback in the source program language, largest possible window into the program, retention of the display as the primary area of interest, and extensive control over execution speed. The appearance of an intervening "system" is kept to a minimum although



Figure 26. Display Messages When the Speed is at STOP

the sense of interacting directly with the program is slightly reduced. Also the operating instructions which must be memorized have increased.

5.2.2 Keyboard and Display without Light Pen. - The next reduction in hardware capability is eliminating the light pen. All source program execution and most functions work as before, but now all input will come from the keyboard. The disadvantages are a general transfer of attention to the keyboard and away from the program itself, and the clear intervention of a "system" between user and program. In particular, the control over execution speed is moved to the keyboard. The speed line is still in the lower right of the display and looks like Figure 27. Typing a digit from 0 to 8 brightens that section of the speed line and runs the source program at the appropriate speed. The 99 is used to remind the user that it is the extra high speed which removes all delay from execution. Single cycle execution is still natural. When the speed is STOP (0), typing 0 executes one cycle.

Keyboard input always brings out the problem that a single character or a short code is not meaningful enough to keep the inexperienced user happy, but a mnemonic code of four or five characters is frustrating to the expert who must type these codes over and over. When messages appear on the screen each message is accompanied by a letter which is the code for the function, as in Figure 28. One types the function code followed by line numbers, variable names, values, or editing instructions as appropriate. Carriage return or line feed is still used to terminate input of an indefinite length such as a new line of code, though no terminator is used in conjunction with the speed control or with functions such as \rightarrow whose arguments are always the same number of characters.

Two functions which used information from the light pen have been slightly modified. First, if the variable name typed when VARIABLES is instituted is already on the screen, then it is removed from the screen. Second, rolling is terminated by a carriage return or line feed.

As with all versions of GRAPE, there is no easy way to accidentally destroy several minutes of work. This version still gives a large window into the operation of a program written in a higher level language and still gives the user a feeling of "hands on" program analysis.

5.2.3 Weaker Refresh Display. - The refresh display used by GRAPE does not need many special hardware features. It needs the capability of displaying several hundred characters without serious flicker, yet have a moderately fast phosphor so that changes on the display do not leave smears.

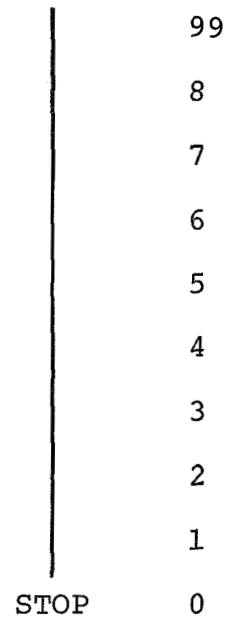


Figure 27. Speed Control

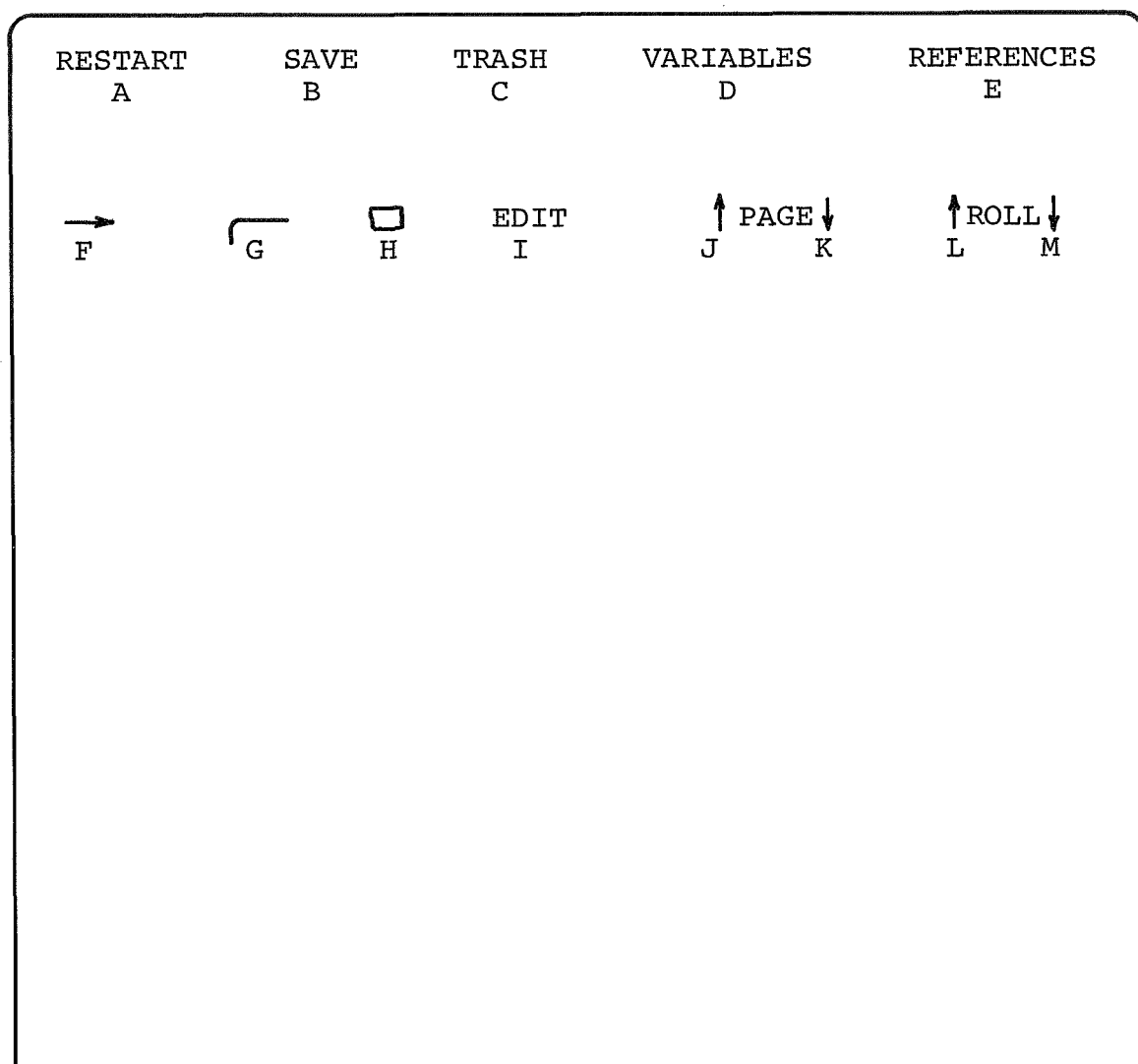


Figure 28. Display Messages When the Speed is at STOP

The mode of production of the characters, whether hardware or software, is not important. Dot capability is necessary only for the feedback associated with the graphical tablets, and one is unlikely ever to find a tablet connected to a display that does not produce dots. If beam intensity variation is not available, brightening some of the program code can be done perfectly well by overwriting the same characters twice or three times each refresh cycle. Other niceties, such as the use of the □ character, can easily be avoided if necessary. The standard display feature of random positioning, as opposed to being obliged to write a single stream of characters which starts at the top left of the screen and ends at the bottom right, is exceedingly useful and is necessary if overwriting is to be done. Display subroutines are not necessary in GRAPE and neither is any curve or vector generator.

5.2.4 Remote (Low Speed) Storage Display. - The most drastic reduction to be considered in the capability of the display is the substitution of a remote storage display for the directly connected refresh display. While the refresh display is completely repainted thirty or more times per second and can be changed just as rapidly, the storage display collects and retains information until it is completely erased and the process restarted. Although some refresh displays are fast enough to produce a typical GRAPE picture in one or two seconds, the use of storage displays which is becoming increasingly common is as low cost computer terminals at the end of telephone lines. Standard telephone lines transmit information at such a rate that it would take ten seconds to produce a page of program code. Thus the standard mode of GRAPE program execution is not suitable for remote storage displays. However, these devices provide visual information remotely from a time shared computer. They are inexpensive and they do not make the computational demand on the computer that a refresh display does. The growing and well deserved popularity of storage displays requires that we consider a way to use them for program analysis.

Keyboard operation for speed control and the institution of execution and editing functions are the same as with the refresh display and keyboard. But for the display of source program execution certain restrictions are apparent. First, it must be possible to indicate the execution of many statements without rewriting the display. Second, displaying as many lines as possible is advantageous since transferring to a new page of program code demands rewriting the display. Third, interaction will be from the keyboard as it was in 5.2.2, but not every keyboard command should cause a rewrite.

Because this is a storage display each statement execution will have to add to the contents of the display face. Indicating

the results of each statement directly on the statement or directly beside the statement will not work for several reasons. When execution is stopped there is no way for the user to tell what statement was last executing. Also some statements need a lot more room than others to display execution. Finally, a statement in a loop could be run dozens of times before one wanted to rewrite the display. Nor can the program code be shifted down one line during the execution of expressions. Therefore, a fairly large display face is necessary to display all the code plus additional material for every execution cycle. The Tektronix cathode ray tube used in several commercially available displays (ref. 19) is 8 1/4" by 6 3/8" and will hold four thousand legible characters.

Since the results of execution cannot be displayed near the statement being executed without greatly reducing the number of statements that can be displayed, and since it is desirable to display as many statements as possible, the display face is arranged with the source statements packed in tightly at the top of the screen. The results of execution are indicated at the bottom of the screen. To perform the packing, unnecessary spaces are removed from the displayed code. The code is arranged in columns as shown in Figure 29 with the statement continuations inserted automatically by the system. The width of each column ensures that most of the executable statements in the particular higher level language will fit on one line commensurate with the number of characters which can be printed on one horizontal line of the display. If three columns of code can be displayed on the screen, then sixty source program statements could be displayed on the top third of the screen.

Just below the displayed statements is a status line. The character on this line indicates whether execution is presently occurring: G for GO and S for STOP. When the status changes the old character is blocked out and a new one is written. Thus if one starts execution by typing 4 (for speed 4), then stops it by typing 0, the status line would go

```
from          STATUS:  S
to            STATUS:  S G
to            STATUS:  S G S
```

A similar method is used for displayed variables. The first variable requested goes on the bottom line of the screen. Whenever its value changes, the old value is blocked out and the new value is written. A second requested variable is displayed on the second from bottom line. If a variable changes enough times to fill a line on the display, then the variable name itself is blocked out and the next available line at the bottom of the screen is used to continue display of that variable.

X means Continuation

| | | |
|---|---|---|
| <div style="position: absolute; left: -50px; top: 50%; transform: translateY(-50%);"><i>Program Code</i></div> 103 _____ 104 _____ 105 _____ X _____ _____ 120 _____ | <div style="position: absolute; left: -50px; top: 50%; transform: translateY(-50%);"><i>Status</i></div> 121 _____ 122 _____ X _____ _____ _____ 146 _____ | <div style="position: absolute; left: -50px; top: 50%; transform: translateY(-50%);"><i>Execution Results</i></div> 147 <input type="checkbox"/> _____ 160 _____ 161 _____ _____ X _____ _____ |
| <div style="position: absolute; left: -50px; top: 50%; transform: translateY(-50%);"><i>Keyboard Input</i></div> STATUS: 88888 | | |
| <div style="position: absolute; left: -50px; top: 50%; transform: translateY(-50%);"><i>Variable</i></div> <div style="text-align: center;"> <p>111 40 110 1x6x2x7 110 1x11 120 70</p> <p>121 FROM 121 20 112 PAUSE</p> <p>CONDENSE 111 122</p> <p>START 111</p> <p>HEIGHT: 2000 04</p> <p>WEIGHT: 2214 21021102 4162102</p> </div> | | |

Figure 29. Display Setup

| | | |
|--------|---|----------|
| Height | : | 19 |
| Weight | : | 36 |
| XXXXXX | : | XX XX XX |

Below the status line is the area for the display of statement executions. The display for each statement is blocked out when the following statement is displayed. Display of execution goes across the screen from left to right and then starts on the left of the next line. The material displayed is the line number followed by the same temporary code that was described in Section II. Understanding the execution will take a little effort on the user's part because the results of execution are displayed at some distance from the source code and because there is no vertical alignment during expression evaluation. When the display of execution cycles fills the open area then the display is rewritten. The size of the open area depends on the number of variables being displayed, and it might be that a program change to a displayed variable is what fills up the screen and causes the display rewrite.

The open area in the center of the screen is also used when execution or editing functions are input from the keyboard. The display is not rewritten after each function is typed. Rather they are listed in the open area as they are typed. The functions all occur when the user requests that they occur or when the open area is filled. One might type

```

DELETE      30
CONDENSE   12 50
START       10

```

before requesting that these functions take effect. In the above example, lines 12, 50, 10 refer to the line numbers displayed when the instructions were typed, even though these numbers might be changed by earlier instructions in the list. The way to request the actual occurrence of the functions in the list is what one might expect: type 0 for single cycle execution.

In the interest of saving space and thus on the storage display saving time, all statements which contain expressions will start

out condensed. Also, no messages are displayed on the screen when execution is stopped. The user must keep a list of the functions and their abbreviations or he must memorize them. Figure 28 is appropriate for the storage display as well as for the refresh display with keyboard input only, except for one set of changes. ROLL no longer exists and the request to PAGE may optionally be followed by the number of pages to be turned in a single rewrite. PAGE replaces ROLL during display output and during the display of multi-valued arrays. The abbreviations from Figure 28 are what the user types but the full words appear on the screen.

The sacrifices which have been made in order to save display rewrite time cause a user some difficulty in that the print is small and the feedback occurs away from the statement. Also, a user might refrain from making some changes because of the time it takes to rewrite the screen. What GRAPE on the remote storage display retains is the idea of the user seeing his program execute and getting a lot of visual feedback in the language of the source program. His window into the program remains large and his control over the execution speed is still extensive. Because the number of separate GRAPE functions is low, it remains an easy system to learn and to use. All three types of program analysis: teaching, debugging, and understanding, are enhanced by the system.

5.3 Conclusions

The design for a system of graphical aids for computer program analysis has been thoroughly worked out. Many of the features have been redesigned several times in order that the final system be as invisible to the user, that is as simple and natural to use, as possible while remaining a complete system with no major desirable features left out.

The rapid increase in the use of computer driven displays and the continued popularity of higher level languages ensures the usefulness of such a system. This system can be used with a wide variety of graphical input/output devices.

That editing information on a display face could be done with a graphical input tablet, a light pen and keyboard, or with a keyboard alone, has already been amply demonstrated. The particular questions which had to be answered by an implementation were whether the dynamic execution of the source program on the display screen did indeed contribute to an understanding of the program and whether the control over execution as provided by the execution functions was sufficiently broad and sufficiently natural. The reactions of the fifty or so people who used or were exposed to the GRAPE system were favorable. GRAPE seems to provide the capabilities one would like when at a desk simulating the

operation of a computer, whether the program is one's own or another's, or whether one is an expert or a novice programmer.

Turning GRAPE into a production system at a particular computer installation involves programming according to the description in Section II. It may also involve two other programming efforts. First is the preparation of the compiler for the higher level language. Today's compilers work only on the single computer for which they were written. And few machines already have incremental compilers for higher level languages. Even if a compiler already exists, GRAPE wants to use it in a special way in order to get both the intermediate code and the final code. This could involve partial rewriting of the compiler.

The second programming effort which may be necessary is the programming of the peripheral devices. Many computer installations which have graphical input/output devices do not yet have suitable software support for them. Each user must write his own input/output package. Although the requirements for this package are clear, learning to program a display can be almost as time consuming as learning to program another computer.

Further research is warranted in the area of GRAPE and low speed storage displays. An implementation is necessary before one can be sure of the display techniques to be used with that device. One could, for example, pre-run a section of code to determine how much of the source program should be put on the screen each time it is rewritten. Further research is also warranted in the application of graphical aids during program execution to the concept of flowchart programming.

REFERENCES

1. QUIKTRAN Reference Manual: International Business Machines Corp., Form J20-0017, San Jose, California, 1967.
2. Lewin, Morton H.: An Introduction to Computer Graphic Terminals. Proc. IEEE, Vol. 55, pp. 1544-1552, 1967.
3. Davis, M. R.; and Ellis, T. O.: The RAND Tablet: A Man-machine Communication Device. Proc. Fall Joint Computer Conference, Vol. 26, pp. 325-331, 1964.
4. Roberts, Lawrence G.: The Lincoln Wand. Proc. Fall Joint Computer Conference, Vol. 29, pp. 223-227, 1966.
5. Richards, Martin: BCPL Reference Manual. Massachusetts Institute of Technology, Project MAC Memorandum M-352, 1968.
6. McCracken, Daniel D.: A Guide to Fortran IV Programming. John Wiley and Sons, New York, 1967.
7. McCracken, Daniel D.: A Guide to Algol Programming. John Wiley and Sons, New York, 1962.
8. Anderson, Robert H.: Syntax-Directed Recognition of Hand-Printed Two-Dimensional Mathematics. Harvard University Doctoral Thesis, 1968.
9. The Compatible Time-Sharing System, A Programmer's Guide. Second Edition, Massachusetts Institute of Technology Press, Cambridge, Mass., 1965.
10. Ide; Munson; Duda; Hurley; et al: "Session on Hand-Printed Character Recognition." Proc. Fall Joint Computer Conference, Vol. 33, pp. 1117-1161, 1968.
11. Corbato, F. J.; and Vyssotsky, V. A.: Introduction and Overview of the Multics System. Proc. Fall Joint Computer Conference, Vol. 27, pp. 185-196, 1965.
12. Bernstein, M. I.: Hand-Printed Input for On-Line Systems. Systems Development Corporation Report TM-3937, Santa Monica, California, 1968.
13. Stotz, Robert: Man-Machine Console Facilities for Computer-Aided Design. Proc. Spring Joint Computer Conference, Vol. 23, pp. 323-328, 1963.

14. Sutherland, Ivan E.: A Head Mounted Three Dimensional Display. Proc. Fall Joint Computer Conference, Vol. 33, pp. 757-764, 1968.
15. Noll, A. Michael: A Computer Technique for Displaying n-Dimensional Hyperobjects. Communications of the Association for Computing Machinery, Vol. 10, pp. 469-473, 1967.
16. Sutherland, William R.: The On-Line Graphical Specification of Computer Procedures. Massachusetts Institute of Technology Doctoral Thesis, 1966.
17. Christensen, C.: An Example of the Manipulation of Directed Graphs in the AMBIT/G Programming Language. Proc. Symposium on Interactive Systems for Experimental Applied Mathematics, Washington, D. C., 1967.
18. QED Reference Manual. Dial-Data, Inc., Newton, Massachusetts, 1968.
19. Advanced Remote Display Station Reference Manual. Computer Displays Inc., Waltham, Massachusetts, 1968.

GENERAL REFERENCES

1. Brady, Paul T.: Writing an Online [Machine Language] Debugging Program for the Experienced User. Communications of the Association for Computing Machinery, Vol. 11, pp. 423-427, 1968.
2. Evans, T. G.; and Darley, D. L.: On-Line Debugging Techniques: A Survey. Proc. Fall Joint Computer Conference, Vol. 29, pp. 37-49, 1966.
3. Lampson, Butler W.: Interactive Machine Language Programming. Proc. Fall Joint Computer Conference, Vol. 27, pp. 473-481, 1965.
4. Schwartz, Jules I.: Online Programming. Communications of the Association for Computing Machinery, Vol. 9, pp. 199-203, 1966.
5. Stockham, Thomas G.: Some Methods of Graphical Debugging. Proc. IBM Scientific Computing Symposium on Man-Machine Communication, Watson Research Center, Yorktown Heights, New York, May, 1965.

APPENDIX A

```
C      J. GREEN                      GRAPE
C
COMMON /IDICOM/ LPNDIM,LPNREG,BRIGHT,INTON,INTOFF,CHRMOD,VECMOD,
X      NLCR,LXR0,LYR0,LXR100,LXR800,LYR800,LYR900,LXR999,VIY0,VIX0,
X      VDY50,VDX100,VIY500,VIY900
COMMON /RUTCOM/ STPBUT,BRKPUT,ENDPUT,SPFED,BRKFLG,ORGVAL,ENDVAL,
X      ORGLNO,ENDLNO,ORGSPD,ENDSPD,BUTTON(100),SETDSP(30)
COMMON /INPCOM/ NUMST,INPUT(80),STUFF(200,11)
COMMON /EXCOM/ J1,J2,J3,J4,IVIZ,PLACE,DSPLN2,EXLINE,GOTO(4),
X      CGOTO(9,3),PAUSE(4),END(4),READ(4),WRITE(4),NOLGT(9),
X      YESLGT(9),ARTHIF(9,3),VAR(36),DSPVAR(5)
COMMON /EDCOM/ K1,K2,LINE1,LINE2,SCRNSZ,TYPE(200),POINT(200),
X      VIZ(200)
COMMON /SUMCOM/ TEMP(30),SPACES(30),STATE(200,4),STLABL(100),
X      DBUF(200),LBUF(200),LETTER(41)
COMMON /OTHCOM/ TOTST,PRESLN,CHARHT,DSPLIN,SCRNMX,TOPLIN,OLDBUT,
X      BREAK(9),SPDLIN(5),SQUARE(13),LINNUM(100)
COMMON /SUBCOM/ SUBRET
COMMON /ARRCOM/ ZARRAY(4,20,10)
```

INITIALIZATION

```
C
C
C
C      CALL DSPATT
      CALL DSPTRN( 1)
      CALL LPNSBL( 1)
      CALL SETCON
      CALL SETLET
      CALL SETARR
      CALL SETRUT
      CALL INSERT
      END
```

```
C
C
C      SUBROUTINE SETCON
      CHARHT=64
      IBRT=12
      IREG=8
      IDIM=4
      ICSIZE=1024+2048
      LPN=256
      SCRNMX=10
      SCRNSZ = 1
      TOTST = 1
      EXLINE = 1
      SUBRET=1
      K1=1
      K2=9999
      LINE1=1
      BRKFLG=0
      TOPLIN=1
      PRESLN=1
      LPNDIM= -20480+16+LPN+IDIM+ICSIZE
      LPNREG= -20480+16+LPN+IREG+ICSIZE
      BRIGHT= -20480+16+IBRT+ICSIZE
      INTON= -20480+16+IDIM+ICSIZE
      INTOFF= -20480+16+ICSIZE
      CHRMOD= -14080
      VECMOD= -14336
      NLCR=2573
      LXR0= -32767-1
      LYR0= -28672
      LXR100= LXR0+80
      LXR800= LXR0+770
      LYR800= LYR0+600
      LYR900= LYR0+900
      LXR999= LXR0+820
      VIY0= 5120
      VIX0= 7168
      VDY50= 4096+50
      VDX100= 6144+100
```

```

VIY500= VIY0+500
VIY900= VIY0+900
RETURN
END

C
C
SUBROUTINE SETLET
DO 410 I=65,90
410 LETTER (I-64)=I
DO 420 I=48,57
420 LETTER (I-21)=I
LETTER (37)=61
LETTER (38)=32
LETTER (39)=43
LETTER (40)=45
LETTER (41)=42
R=32*256
S=32
T=R+S
DO 430 I=1,4
READ(I)=T
WRITE(I)=T
PAUSE(I)=T
GOTO(I)=T
430 END(I)=T
DO 440 I=1,30
440 SPACES(I)=T
DO 450 I=1,9
NOLGT(I)=T
YESLGT(I)=T
DO 450 J=1,3
ARTHIF(I,J)=T
450 CGOTO(I,J)=T
NOLGT(3)=40+R
NOLGT(7)=41*256+S
YESLGT(7)=L(7)+R
YESLGT(8)=L(15)*256+S
YESLGT(9)=L(20)*256+L(15)
READ(2)=R+L(18)
READ(3)=L(5)*256+L(1)
READ(4)=L(4)*256+S
GOTO(2)=YESLGT(7)
GOTO(3)=YESLGT(8)
GOTO(4)=YESLGT(9)
WRITE(2)=R+L(23)
WRITE(3)=L(18)*256+L(9)
WRITE(4)=L(20)*256+L(5)
PAUSE(2)=R+L(16)
PAUSE(3)=L(1)*256+L(21)
PAUSE(4)=L(19)*256+L(5)
END(2)=R+L(5)
END(3)=L(14)*256+L(4)
CGOTO(6,1)=24415
CGOTO(7,2)=95+R
CGOTO(8,2)=24320
CGOTO(9,3)=24415
ARTHIF(5,1)=24415
ARTHIF(6,2)=95+R
ARTHIF(7,2)=24320
ARTHIF(8,3)=24415
RETURN
END

C
C
SUBROUTINE SETARR
DO 50 I=1,30
VIZ(I)=0
TYPE(I)=0
DBUF(I)=0
LBUF(I)=0
POINT(I)=0
DO 30 J=1,4
30 STATE(I,J)=0
DO 40 J=1,11

```



```

40  STUFF(J,I)=0
50  CONTINUE
    DO 100 I=1,100
      LINNUM(I)=0
      BUTT0N(I)=0
100  STLABL(I)=0
    DO 110 I=1,26
110  VAR(I)=0
    DO 120 I=27,36
120  VAR(I)=I-27
    DO 125 I=1,4
      DO 125 J=1,20
        DO 125 K=1,10
125  ZARRAY(I,J,K)=0
    DO 130 I=1,30
      TEMP(I)=0
130  SETDSP(I)=0
    DO 140 I=1,5
140  DSPVAR(I)=0
      VIZ(1) = 1
      TYPE(1) = 1
      LINNUM(1)=1
      IT=32*256
      STUFF(1,1)=LETTER(3)
      STUFF(2,1)=LETTER(38)
      STUFF(3,1)=LETTER(38)
      STUFF(4,1)=LETTER(10)
      STUFF(5,1)=LETTER(8)
      STUFF(6,1)=LETTER(7)
      BREAK(1)=CHRM0D
      BREAK(2)=LXR100
      BREAK(3)=INTON
      BREAK(4)=0
      BREAK(5)=IT+32
      BREAK(6)=126*256+32
      SETDSP(1)=BRIGHT
      SETDSP(2)=LYR0
      SETDSP(3)=LXR100
      SETDSP(4)=CHRM0D
      SPDLIN(1)= INTON
      SPDLIN(2)= VECMOD
      SPDLIN(3)= LXR999
      SPDLIN(4)= VIY500
      SPDLIN(5)= VDY50
      SQUARE(1)=CHRM0D
      SQUARE(2)=LXR100
      SQUARE(3)=INTON
      SQUARE(4)= 0
      SQUARE(5)=IT+32
      SQUARE(6)=35*256+32
      SQUARE(7)=NLCR
      RETURN
    END
C
C
C  SUBROUTINE SETBUT
C  NUMBUT=0
C
C  LINE NUMBERS
C  ORGLNO=NUMBUT
  TEMP(1)=INTON
  TEMP(2)=VECMOD
  TEMP(3)=LXR0
  TEMP(4)=VIY900
  TEMP(5)=CHRM0D
  CALL MOVU2D(TEMP,0,5)
  DO 50 I=1,SCRNMX
    NUMBUT=NUMBUT+1
    CALL DSPSIZ(BUTT0N(NUMBUT))
    CALL BN2DEC(I,TEMP(1))
    TEMP(1)=INTON
    TEMP(2)=NLCR
50  CALL MOVU2D(TEMP,0,3)
    ENDLNO=NUMBUT

```

```

C
C   SPEED LINE
C   CALL MOVU2D(SPDLIN,0,5)
C
C   SPEED DOTS
C   ORGSPD=NUMBUT+1
C   TEMP(1)=LPNDIM
C   TEMP(3)=0
C   DO 60 I=50,500,50
C   NUMBUT=NUMBUT+1
C   CALL DSPSIZ(BUTTON(NUMBUT))
C   TEMP(2)=I+VIY0
60 CALL MOVU2D(TEMP,0,3)
C   ENDSPD=NUMBUT
C
C   WORD STOP
C   NUMBUT=NUMBUT+1
C   STPBUT=NUMBUT
C   CALL DSPSIZ(BUTTON(NUMBUT))
C   TEMP(1)=BRIGHT
C   TEMP(2)=VECMOD
C   TEMP(3)=LXR800
C   TEMP(4)=VIY0
C   TEMP(5)=CHRMOD
C   TEMP(6)=LETTER(19)
C   TEMP(7)=LETTER(20)
C   TEMP(8)=LETTER(15)
C   TEMP(9)=LETTER(16)
C   CALL MOVU2D(TEMP,0,9)
C
C   BREAK LINE
C   NUMBUT=NUMBUT+1
C   BRKBUT=NUMBUT
C   CALL DSPSIZ(BUTTON(NUMBUT))
C   CALL MOVU2D(BREAK,0,6)
C   CALL MOVU2D(INTOFF,BUTTON(BRKBUT)+2,1)
C
C   DSPLIN+DSPLN2
C   CALL DSPSIZ(DSPLIN)
C   CALL MOVU2D(SETDSP,0,30)
C   CALL DSPSIZ(DSPLN2)
C   CALL MOVU2D(SETDSP,0,30)
C   PLACE = DSPLIN+4
C
C   DISPLAYED VALUES
C   ORGVAL=NUMBUT
C   TEMP(1)=INTOFF
C   TEMP(2)=LYR800
C   TEMP(3)=LXR800
C   TEMP(4)=CHRMOD
C   DO 80 I=1,5
C   NUMBUT=NUMBUT+1
C   CALL DSPSIZ(BUTTON(NUMBUT))
C   CALL MOVU2D(TEMP,0,4)
C   CALL MOVU2D(SPACES,0,20)
80 TEMP(2)=TEMP(2)+CHARHT
C   ENDVAL=NUMBUT
C
C   EDIT FUNCTIONS
C   FINISH
C   OLDBUT=ORGLNO+1
C   SPEED=STPBUT
C   TEMP(1)=INTON
C   TEMP(2)=LXR100
C   TEMP(3)=LYR900-CHARHT
C   TEMP(4)=CHRMOD
C   CALL MOVU2D(TEMP,0,4)
C   NUMBUT=NUMBUT+1
C   ENDBUT=NUMBUT
C   CALL DSPSIZ(BUTTON(NUMBUT))
C   RETURN
C
C   END

```

COMPILATION

```
C
C
SUBROUTINE READST (LINE)
NUMST = LINE
IF(K2.LE.1)GO TO 40
CALL RDIN(INPUT(41))
DO 30 I=1,40
INPUT(I+40)=INPUT(I+40)-129-32767
INPUT(2*I-1)=INPUT(I+40)/256
30 INPUT(2*I)=INPUT(I+40)-INPUT(2*I-1)*256
GO TO 70
40 CALL TTYENB
DO 50 I=1,80
CALL TIIN(J)
INPUT(I)=J
C
C CHECK FOR LINE FEED
IF(INPUT(1).EQ.13) GO TO 70
50 CONTINUE
70 DO 80 J=1,80
80 INPUT(J)=0
DO 85 I=6,80
85 INPUT(I-3)=INPUT(I)
C
C CHECK FOR DOLLAR SIGN
IF(INPUT(1).EQ.36) RETURN
VIZ(NUMST)=1
90 IF(INPUT(1).NE.LETTER(3)) GO TO 100
C
C COMMENT
TYPE(NUMST)=1
CALL PACK
RETURN
100 IF(INPUT(1).EQ.BLANK) GO TO 120
CALL STNUM(1,S)
STLABL(S)=NUMST
120 I=INPUT(5)
IF(INPUT(12).EQ.EQUAL) GO TO 250
IF(INPUT(6).EQ.LETTER(26)) GO TO 260
IF(I.EQ.LETTER(1).AND.INPUT(4).EQ.LETTER(3)) GO TO 270
IF(I.EQ.LETTER(21)) GO TO 280
IF(I.EQ.LETTER(5).AND.INPUT(6).EQ.LETTER(20)) GO TO 290
IF(I.EQ.LETTER(1)) GO TO 130
IF(I.EQ.LETTER(5)) GO TO 140
IF(I.EQ.LETTER(6)) GO TO 160
IF(I.EQ.LETTER(14)) GO TO 180
IF(I.EQ.LETTER(15)) GO TO 190
IF(I.EQ.LETTER(18)) GO TO 210
IF(I.EQ.EQUAL) GO TO 220
GO TO 999
C
C PAUSE
130 TYPE(NUMST)=4
CALL PACK
RETURN
C
C READ
140 TYPE(NUMST)=6
CALL VARIAB(9,1)
CALL PACK
RETURN
C
C IF
160 IF(INPUT(20).EQ.0 .OR. INPUT(20).EQ.BLANK) GO TO 170
C
C LOGICAL IF
CALL VARIAB(7,1)
CALL VARIAB(12,2)
CALL STNUM(20,S)
STATE(NUMST,3)=S
CALL PACK
IF(INPUT(9).EQ.LETTER(5)) TYPE(NUMST)=22
IF(INPUT(9).EQ.LETTER(14)) TYPE(NUMST)=23
```

```

        IF (INPUT(9).EQ.LETTER(12).AND.INPUT(10).EQ.LETTER(20))
X      TYPE(NUMST)=24
        IF (INPUT(9).EQ.LETTER(12).AND.INPUT(10).EQ.LETTER(5))
X      TYPE(NUMST)=25
        IF (INPUT(9).EQ.LETTER(7).AND.INPUT(10).EQ.LETTER(20))
X      TYPE(NUMST)=26
        IF (INPUT(9).EQ.LETTER(7).AND.INPUT(10).EQ.LETTER(5))
X      TYPE(NUMST)=27
        RETURN
C
C      ARITHMETIC IF
170  TYPE(NUMST)=21
        CALL VARIAB(7,1)
        CALL STNUM(9,S)
        STATE(NUMST,2)=S
        CALL STNUM(12,S)
        STATE(NUMST,3)=S
        CALL STNUM(15,S)
        STATE(NUMST,4)=S
        CALL PACK
        RETURN
C
C      END
180  TYPE(NUMST)=5
        CALL PACK
        RETURN
C
C      GO TO
190  IF (INPUT(15).NE.0 .AND. INPUT(15).NE.BLANK) GO TO 200
C
C      GO TO 55
        TYPE(NUMST)=2
        CALL STNUM(10,S)
        STATE(NUMST,1)=S
        CALL PACK
        RETURN
C
C      COMPUTED GO TO
200  TYPE(NUMST)=3
        CALL STNUM(11,S)
        STATE(NUMST,1)=S
        CALL STNUM(14,S)
        STATE(NUMST,2)=S
        CALL STNUM(17,S)
        STATE(NUMST,3)=S
        CALL VARIAB(21,4)
        CALL PACK
        RETURN
C
C      WRITE
210  TYPE(NUMST)=7
        CALL VARIAB(10,1)
        CALL PACK
        RETURN
C
C      ASSIGNMENT
220  CALL VARIAB(4,1)
        CALL VARIAB(6,2)
        OPONE=INPUT(7)
        IF (OPONE.NE.0 .AND. OPONE.NE.BLANK) GO TO 230
        TYPE(NUMST)=8
        CALL PACK
        RETURN
230  CALL VARIAB(8,3)
        IF (OPONE.EQ.PLUS) TYPE(NUMST)=9
        IF (OPONE.EQ.MINUS) TYPE(NUMST)=13
        IF (OPONE.EQ.STAR) TYPE(NUMST)=17
        OPTWO=INPUT(9)
        IF (OPTWO.NE.0 .AND. OPTWO.NE.BLANK) GO TO 240
        CALL PACK
        RETURN
240  CALL VARIAB(10,4)
        IF (OPTWO.EQ.PLUS) TYPE(NUMST)=TYPE(NUMST)+1
        IF (OPTWO.EQ.MINUS) TYPE(NUMST)=TYPE(NUMST)+2

```

```

      IF(OPTWO.EQ.STAR) TYPE(NUMST)=TYPE(NUMST)+3
      CALL PACK
      RETURN
C
C      ARRAY=VARIABLE
250  TYPE(NUMST)=28
      CALL VARIAB(6,1)
      CALL VARIAB(8,2)
      CALL VARIAB(10,3)
      CALL VARIAB(13,4)
      CALL PACK
      RETURN
C
C      VARIABLE = ARRAY
260  TYPE(NUMST)=29
      CALL VARIAB(4,1)
      CALL VARIAB(8,2)
      CALL VARIAB(10,3)
      CALL VARIAB(12,4)
      CALL PACK
      RETURN
C
C      CALL
270  TYPE(NUMST)=30
      CALL STNUM(16,S)
      STATE(NUMST,1)=S
      CALL PACK
      RETURN
C
C      SUBROUTINE
280  TYPE(NUMST)=31
      CALL PACK
      RETURN
C
C      RETURN
290  TYPE(NUMST)=32
      CALL PACK
      RETURN
C
999  CALL EDERR(14)
      END
C
C
      SUBROUTINE STNUM (M,N)
      DO 110 I=1,2
      TEMP(I)=100
      L=I+M-1
      DO 110 J=27,36
      IF ( INPUT(L).EQ.LETTER(J)) TEMP(I)=J-27
110  CONTINUE
      IF(TEMP(1)*TEMP(2).GT.90.OR.TEMP(1)+TEMP(2).EQ.0) CALL EDERR(30)
      N=TEMP(1) * 10 + TEMP(2)
      RETURN
      END
C
C
      SUBROUTINE VARIAB(I,J)
      DO 20 K=1,36
20  IF(INPUT(I) .EQ. LETTER(K)) STATE(NUMST,J)=K
      RETURN
      END
C
C
      SUBROUTINE PACK
      J=0
      DO 10 I=1,21,2
      J=J+1
10  STUFF(J,NUMST)=INPUT(I) * 256 + INPUT(I+1)
      RETURN
      END
C

```

EXECUTION

```

C      SUBROUTINE XECUTE (LINE)
C

```

```

        IF(LINE.EQ.0)CALL EXERR(20)
        EXLINE = LINE
        IF (LPNCHK( 1).GT.1) CALL PENSEE
        J1=STATE(EXLINE,1)
        J2=STATE(EXLINE,2)
        J3=STATE(EXLINE,3)
        J4=STATE(EXLINE,4)
        IVIZ=VIZ(EXLINE)
        ITYPE=TYPE(EXLINE)
        IF (IVIZ.EQ.1.AND.LBUF(EXLINE).EQ.0) CALL REGEN(EXLINE)
        IF (IVIZ.LT.4) GO TO (1020,1900,1030), IVIZ
1020  GO TO (1030,1070,1080,1090,1100,1110,1120,1130,1130,1130,1130,
X     1130,1130,1130,1130,1130,1130,1130,1130,1130,1260,1270,1270,
X     1270,1270,1270,1270,1280,1290,1300,1310,1320), ITYPE
C
1030  CALL XCOMNT
1070  CALL XGOTO
1080  CALL XCGOTO
1090  CALL XPAUSE
1100  CALL XEND
1110  CALL XREAD
1120  CALL XWRITE
1130  CALL XRITH(ITYPE-7)
1260  CALL XARTHF
1270  CALL XLOGIF(ITYPE-21)
1280  CALL XARREV
1290  CALL XVEARR
1300  CALL XSBCAL
1310  CALL XSUBRU
1320  CALL XRETRN
1900  IF (BRKFLG.EQ.0) CALL BRAKE
        BRKFLG = 0
        GO TO 1020
        END
C
C
        SUBROUTINE PENSEE
        INTEGER PENBUF(50)
        CALL LPNRD( PENBUF)
        I=IABS(PENBUF(2))
        CALL LPNCLR
        IF (I.GT.BUTTON(STPRUT).AND.I.LT.BUTTON(STPBUT+1)) CALL STOOP
        CALL MOVU2D (INTOFF,BUTTON(BRKBIT)+2,1)
        SETDSP(2)=LYR0
        CALL MOVU2D(SETDSP,DSPLN2,30)
        DO 30 J = ORGSPD,ENDSPD
        IF (I.LT.BUTTON(J).OR.I.GT.BUTTON(J+1)) GO TO 30
        CALL MOVU2D (LPNDIM,BUTTON(SPEED),1)
        SPEED = J
        CALL MOVU2D (BRIGHT,BUTTON(SPEED),1)
        RETURN
30    CONTINUE
        RETURN
        END
C
C
        SUBROUTINE XCOMNT
        IF(IVIZ.GT.2) GO TO 20
        CALL UNDO
        CALL MOVU2D(LETTER(3),PLACE,1)
        CALL DELAY
20    CALL XECUTE(POINT(EXLINE))
        END
C
C
        SUBROUTINE XGOTO
        IF(IVIZ .GT. 2 ) GO TO 1074
        CALL UNDO
        CALL MOVU2D(GOTO,PLACE,4)
        CALL DELAY
1074  IF(STLABL(J1) .EQ. 0) CALL EXERR(16)
        CALL XECUTE(STLABL(J1))
        END
C

```

```

C
SUBROUTINE XCGOTO
I=VAR(J4)
IF(RANGE(I,1,3)) CALL EXERR(2)
J1=STATE(EXLINE,I)
IF(IVIZ .GT. 2 ) GO TO 100
CALL UNDO
CALL MOVU2D(CGOTO(1,I),PLACE,10)
CALL DELAY
100 IF(STLABL(J1) .EQ. 0) CALL EXERR(17)
CALL XECUTE(STLABL(J1))
END

C
SUBROUTINE XPAUSE
IF(IVIZ .GT. 2 ) GO TO 1092
SPOT=PLACE
CALL UNDO
GO TO 1091
1092 CALL MOVU2D(BRIGHT,DSPLN2,1)
SPOT=DSPLN2+3
1091 CALL MOVU2D(PAUSE,SPOT,4)
EXLINE=POINT(EXLINE)
CALL STOOP
END

C
SUBROUTINE XEND
IF(IVIZ .GT. 2 ) GO TO 1105
CALL UNDO
SPOT=PLACE
GO TO 1102
1105 CALL MOVU2D(BRIGHT,DSPLN2,1)
SPOT=DSPLN2+3
1102 CALL MOVU2D(END,SPOT,3)
CALL STOOP
END

C
SUBROUTINE XREAD
IF(IVIZ .GT. 2 ) GO TO 1115
CALL UNDO
CALL MOVU2D(READ,PLACE,4)
READ(1,1111) VAR(J1)
1111 FORMAT(I5)
GO TO 10
1115 CALL MOVU2D(STUFF(1,EXLINE),DSPLN2+3,11)
CALL MOVU2D(BRIGHT,DSPLN2,1)
READ(1,1111) VAR(J1)
CALL MOVU2D(INTOFF,DSPLN2,1)
10 DO 20 I=1,5
IF(DSPVAR(I).EQ.J1) CALL PUTVAR(I,J1)
20 CONTINUE
CALL XECUTE(POINT(EXLINE))
END

C
SUBROUTINE XWRITE
IF(IVIZ .GT. 2 ) GO TO 1125
CALL UNDO
CALL MOVU2D(WRITE,PLACE,5)
WRITE(1,1121) VAR(J1)
1121 FORMAT(/I6/)
CALL DELAY
GO TO 10
1125 CALL MOVU2D(STUFF(1,EXLINE),DSPLN2+3,11)
CALL MOVU2D(BRIGHT,DSPLN2,1)
WRITE(1,1121) VAR(J1)
CALL MOVU2D(INTOFF,DSPLN2,1)
10 CALL XECUTE(POINT(EXLINE))
END

C
SUBROUTINE XRITH(N)

```

```

        GO TO (1130,1140,1150,1160,1170,1180,1190,1200,
X 1210,1220,1230,1240,1250), N
1130  VAR(J1)=VAR(J2)
      GO TO 3000
1140  IDUM=  VAR(J3)
      GO TO 2970
1150  IDUM=  VAR(J3)+VAR(J4)
      GO TO 2970
1160  IDUM=  VAR(J3)-VAR(J4)
      GO TO 2970
1170  IDUM=  VAR(J3)*VAR(J4)
      GO TO 2970
1180  IDUM=  -VAR(J3)
      GO TO 2970
1190  IDUM=  VAR(J4)-VAR(J3)
      GO TO 2970
1200  IDUM=  -VAR(J3)-VAR(J4)
      GO TO 2970
1210  IDUM=  -VAR(J3)*VAR(J4)
      GO TO 2970
1220  VAR(J1)=VAR(J2)*VAR(J3)
      GO TO 3000
1230  VAR(J1)=VAR(J2)*VAR(J3)+VAR(J4)
      GO TO 3000
1240  VAR(J1)=VAR(J2)*VAR(J3)-VAR(J4)
      GO TO 3000
1250  VAR(J1)=VAR(J2)*VAR(J3)*VAR(J4)
      GO TO 3000
2970  VAR(J1)=IDUM  +VAR(J2)
3000  CALL XARITH
      END
C
C
      SUBROUTINE XARTHF
      IF(VAR(J1)) 1262,1264,1266
1262  JDUM=1
      GO TO 3100
1264  JDUM=2
      GO TO 3100
1266  JDUM=3
3100  J1=STATE(EXLINE,JDUM+1)
      IF(IVIZ .GT. 2 ) GO TO 3200
      CALL UNDO
      CALL MOVU2D(ARTHF(1,JDUM),PLACE,8)
      CALL DELAY
3200  IF(STLABL(J1) .EQ. 0) CALL EXERR(1)
      CALL XECUTE(STLABL(J1))
      END
C
C
      SUBROUTINE XARITH
      DO 3010 I=1,5
      IF(DSPVAR(I).NE.J1) GO TO 3010
      CALL PUTVAR(I,J1)
      GO TO 3020
3010  CONTINUE
3020  IF(IVIZ .GT. 2 ) GO TO 3030
      CALL UNDO
      CALL MOVU2D(SPACES,PLACE,12)
      TEMP(1)=EQUAL
      CALL BN2DEC(VAR(J1),TEMP(2))
      CALL MOVU2D(TEMP,PLACE+5,4)
      CALL DELAY
3030  CALL XECUTE(POINT(EXLINE))
      END
C
C
      SUBROUTINE XLOGIF (N)
      GO TO (1270,1280,1290,1300,1310,1320), N
1270  IF(VAR(J1).EQ.VAR(J2)) GO TO 1340
      GO TO 1330
1280  IF(VAR(J1).NE.VAR(J2)) GO TO 1340
      GO TO 1330

```



```

1290 IF(VAR(J1).LT.VAR(J2)) GO TO 1340
      GO TO 1330
1300 IF(VAR(J1).LE.VAR(J2)) GO TO 1340
      GO TO 1330
1310 IF(VAR(J1).GT.VAR(J2)) GO TO 1340
      GO TO 1330
1320 IF(VAR(J1).GE.VAR(J2)) GO TO 1340
      GO TO 1330
1330 CALL XNOLGF
1340 CALL XYSLGF
      END
C
C
      SUBROUTINE XYSLGF
      J1=STATE(EXLINE,3)
      IF(IVIZ.GT. 2 ) GO TO 1350
      CALL UNDO
      CALL MOVU2D(YESLGT,PLACE,9)
      CALL DELAY
1350 IF(STLABL(J1).EQ.0) CALL EXERR(15)
      CALL XECUTE (STLABL(J1))
      END
C
C
      SUBROUTINE XNOLGF
      IF(IVIZ.GT. 2 ) GO TO 10
      CALL UNDO
      CALL MOVU2D(NOLGT,PLACE,7)
      CALL DELAY
10    CALL XECUTE(POINT(EXLINE))
      END
C
C
      SUBROUTINE XSUBRU
      INTEGER SUBR(7)
      DATA SUBR(1),SUBR(2),SUBR(3),SUBR(4),SUBR(5),SUBR(6),SUBR(7)
X    /2H ,2H S,2HUB,2HRO,2HUT,2HIN,1HE/
      DO 10 I=1,7
10    IF(SUBR(I).LT.0)SUBR(I)=SUBR(I)-128-32767-1
      IF(IVIZ.GT.2) GO TO 20
      CALL UNDO
      CALL MOVU2D(SUBR,PLACE,7)
      CALL DELAY
20    CALL XECUTE(POINT(EXLINE))
      END
C
C
      SUBROUTINE XSBCAL
      INTEGER CALL(4)
      DATA CALL(1),CALL(2),CALL(3),CALL(4)/2H ,2H C,2HAL,1HL/
      DO 10 I=1,4
10    IF(CALL(I).LT.0)CALL(I)=CALL(I)-128-32767-1
      IF(IVIZ.GT.2) GO TO 20
      CALL UNDO
      CALL MOVU2D(CALL,PLACE,4)
      CALL DELAY
20    IF(STLABL(J1).EQ.0) CALL EXERR(50)
      SUBRET=EXLINE
      CALL XECUTE(STLABL(J1))
      END
C
C
      SUBROUTINE XRETRN
      INTEGER RETU(5)
      DATA RETU(1),RETU(2),RETU(3),RETU(4),RETU(5)/2H ,2H R,
X    2HET,2HUR,1HN/
      DO 10 I=1,5
10    IF(RETU(I).LT.0)RETU(I)=RETU(I)-128-32767-1
      IF(IVIZ.GT.2) GO TO 20
      CALL UNDO
      CALL MOVU2D(RETU,PLACE,5)
      CALL DELAY
20    CALL XECUTE(POINT(SUBRET))
      END

```

```

C
C
SUBROUTINE XARREV
I=VAR(J1)
J=VAR(J2)
K=VAR(J3)
ZARRAY(I,J,K)=VAR(J4)
J1=J4
CALL XARITH
END

C
C
SUBROUTINE XVEARR
I=VAR(J2)
J=VAR(J3)
K=VAR(J4)
VAR(J1)=ZARRAY(I,J,K)
CALL XARITH
END

C
C
SUBROUTINE BRAKE
TEMP(1) = BRIGHT
TEMP(2) = LYR900-LBUF(EXLINE)*CHARHT
CALL MOVU2D (TEMP,BUTTON(BRKBT)+2,2)
BRKFLG=1
CALL STOOP
END

C
C
SUBROUTINE UNDO
IF (PRESLN.EQ.0) GO TO 50
CALL MOVU2D (INTON ,BUTTON(OLDBUT),1)
CALL MOVU2D (INTON,DBUF(PRESLN),1)
50  OLDBUT = ORGLNO + LBUF (EXLINE)
PRESLN = EXLINE
SETDSP(2) = LYR900-LBUF(EXLINE)*CHARHT
CALL MOVU2D (SETDSP,DSPLIN,25)
CALL MOVU2D (BRIGHT,BUTTON(OLDBUT),1)
RETURN
END

C
C
SUBROUTINE DELAY
K=(ENDSPD-SPEED)*5
DO 20 I=1,K
DO 20 L=1,2
DO 20 J=1,5000
20  CONTINUE
RETURN
END

C
C
SUBROUTINE EXERR(N)
WRITE(1,10)N
10  FORMAT(16H EXECUTION ERROR,I4)
CALL EDITOR
END

C
C
SUBROUTINE EDITOR
40  CALL TTYFNR
50  IF(LPNCHK(1).GT.1) CALL XECUTE(EXLINE)
CALL TTYCHK(FLG,CHARTR)
IF(.NOT.FLG) GO TO 50
READ(1,100) FCN,K1,K2
100  FORMAT(A2,2I2)
DO 150 I=1,16
IF(FCN.EQ.COM(I)) GO TO 300
150  CONTINUE
CALL EDERR(20)
300  IF(K2.EQ.99) GO TO 40
LINE1=LINNUM(K1)
LINE2=LINNUM(K2)

```

FUNCTION

```

GO TO (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20),I
1  CALL CHANGE
2  CALL INSERT
3  CALL DELFTE
4  CALL CONDEN
5  CALL EXPAND
6  CALL START
7  CALL MOVE
8  CALL SETBRK
9  CALL KILBRK
10 CALL VARSET
11 CALL SHOVAR
12 CALL REMVAR
13 CALL SAVE
14 CALL RESTRT
15 CALL PAGE
16 CALL ROLL
17 CALL EDERR(20)
18 CALL EDERR(20)
19 CALL EDERR(20)
20 CALL EDERR(20)
END

C
C
SUBROUTINE STOOP
CALL MOVU2D (LPNDIM,BUTTON(SPEED),1)
SPEED=STPBUT
CALL MOVU2D (BRIGHT,BUTTON(STPBUT),1)
CALL EDITOR
END

C
C
SUBROUTINE REGEN(IJK)
IF (IJK.GT.TOTST) CALL EDERR(36)
IF (IJK.NE.0) TOPLIN=IJK
5  CALL DSPCUT( BUTTON(ENDBUT))
DO 10 I=1,TOTST
LBUF(I)=0
10 DBUF(I)=0
CALL MOVU2D(INTOFF,BUTTON(BRK BUT)+2,1)
CFLAG=1
LINE=TOPLIN
SCRNSZ=0
20 JVIZ=VIZ(LINE)
IF (JVIZ.GT.4) GO TO 80
GO TO (30,30,80,50),JVIZ
30 IF(SCRNSZ.GE.SCRNMX) GO TO 200
SCRNSZ=SCRNSZ+1
GO TO (40,35),JVIZ
35 BREAK(4)=LYR900-SCRNSZ*CHARHT
CALL MOVU2D(BREAK,0,6)
40 CALL DSPSIZ(DBUF(LINE))
LBUF(LINE)=SCRNSZ
LINNUM(SCRNSZ)=LINE
CFLAG=0
TEMP(1)=INTON
TEMP(2)=LXR100
CALL MOVU2D(TEMP,0,2)
CALL MOVU2D(STIFF(1,LINE),0,11)
CALL MOVU2D(NLCR,0,1)
GO TO 80
50 IF(CFLAG.EQ.1)GO TO 80
CFLAG=1
SQUARE(4)=LYR900-SCRNSZ*CHARHT
CALL MOVU2D(SQUARE,0,7)
80 LINE=POINT(LINE)
IF(LINE.NE.0) GO TO 20
200 CALL MOVU2D(INTON,BUTTON(OLDBUT),1)
CALL MOVU2D(INTOFF,DSPLIN,1)
IF(LBUF(PRESLN).EQ.0) RETURN
CALL MOVU2D(LYR900-LBUF(PRESLN)*CHARHT,BUTTON(BRK BUT)+3,1)
OLDBUT=ORGLNO+LBUF(PRESLN)
CALL MOVU2D(BRIGHT,BUTTON(OLDBUT),1)
RETURN
END

```

```

C
C
SUBROUTINE CHANGE
IF(RANGE(K1,1,SCRNSZ))CALL EDERR(1)
CALL READST(LINE1)
CALL REGEN(0)
CALL T1OUT(10)
CALL EDITOR
END

C
C
SUBROUTINE INSERT
IF(RANGE(K1,1,SCRNSZ )) CALL EDERR(2)
DO 20 I=1,K2
CALL READST(TOTST+1)
C
CHECK FOR DOLLAR SIGN
IF(INPUT(1).NE.36) GO TO 5
CALL SEARCH(4,0,1)
GO TO 30
5
TOTST=TOTST+1
DUMMY=LINE1
10
LINE1=DUMMY
DUMMY=POINT(LINE1)
IF(VIZ(DUMMY) .GT.2 .AND.DUMMY.NE.0) GO TO 10
POINT(TOTST)=DUMMY
POINT(LINE1)=TOTST
20
LINE1=TOTST
30
CALL REGEN(0)
CALL T1OUT(10)
CALL EDITOR
END

C
C
SUBROUTINE DELETE
IF(RANGE(K1,1,SCRNSZ)) CALL EDERR(24)
DUMMY=LINE1
133
DUMMY= POINT(DUMMY)
IF(VIZ(DUMMY) .EQ. 3) GO TO 133
IF(VIZ(DUMMY) .GE. 4) CALL EDERR(12)
VIZ(LINE1)=3
CALL REGEN(0)
CALL EDITOR
END

C
C
SUBROUTINE CONDENSE
IF(K1.LE. 0 .OR. K1.GE.K2 .OR.K2 .GT. SCRNSZ) CALL EDERR(27)
CFLAG=0
10
LINE1=POINT(LINE1)
IF(LINE1.EQ.0) GO TO 30
JVIZ=VIZ(LINE1)
IF(JVIZ .LT. 4) GO TO (20,20,10),JVIZ
IF(CFLAG .EQ. 1) VIZ(LINE1)=JVIZ+1
GO TO 10
20
K1=K1+1
IF(K1.GT.K2) GO TO 30
CFLAG=1
VIZ(LINE1)=4
GO TO 10
30
CALL REGEN(0)
CALL EDITOR
END

C
C
SUBROUTINE EXPAND
IF(RANGE(K1,1,SCRNSZ)) CALL EDERR(28)
10
LINE1=POINT(LINE1)
IF(LINE1.EQ.0) GO TO 30
JVIZ=VIZ(LINE1)
IF(JVIZ.LT.5) GO TO (30,30,10,20),JVIZ
VIZ(LINE1)=JVIZ-1
GO TO 10
20
VIZ(LINE1)=1
GO TO 10

```

```

30  CALL REGEN(0)
    CALL EDITOR
    END
C
C
    SUBROUTINE START
    IF (RANGE(K1,1,SCRNSZ)) CALL EDERR(5)
    EXLINE=LINE1
    BRKFLG=0
    IF (VIZ(LINE1).EQ.2) BRKFLG = 1
    CALL UNDO
    CALL REGEN(0)
    CALL EDITOR
    END
C
C
    SUBROUTINE MOVE
    IF (RANGE(K1,1,SCRNSZ)) CALL EDERR(13)
    IF (RANGE(K2,1,SCRNSZ)) CALL EDERR(14)
    IF (LINE1.EQ.1) CALL EDERR(15)
    IF (K1.EQ.K2.OR.K2.EQ.K1-1) CALL EDERR(15)
    DO 10 LINEA = 1,1000
    IF (POINT(LINEA).EQ.LINE1) GO TO 20
    CONTINUE
10  DO 30 LINEB= LINE1,1000
20  I = POINT(LINEB)
    IF (VIZ(I).LE.2.OR.I.EQ.0) GO TO 40
    CONTINUE
30  DO 50 LINEC = LINE2,1000
40  J = POINT(LINEC)
    IF (VIZ(J).LE.2.OR.J.EQ.0) GO TO 60
    CONTINUE
50  POINT (LINEA) = I
60  POINT (LINEB) = J
    POINT (LINEC) = LINE1
    CALL REGEN(0)
    CALL EDITOR
    END
C
C
    SUBROUTINE SETBRK
    IF (RANGE(K1,1,SCRNSZ)) CALL EDERR(3)
    VIZ(LINE1)=2
    CALL REGEN(0)
    CALL EDITOR
    END
C
C
    SUBROUTINE KILBRK
    IF (RANGE(K1,1,SCRNSZ)) CALL EDERR(4)
    VIZ(LINE1)=1
    CALL REGEN(0)
    CALL EDITOR
    END
C
C
    SUBROUTINE VAR$FT
    IF (RANGE(K1,1,26)) CALL EDERR(9)
    VAR(K1)=K2
    DO 10 I=1,5
10  IF (DSPVAR(I) .EQ. K1) CALL PUTVAR(I,K1)
    CONTINUE
    CALL EDITOR
    END
C
C
    SUBROUTINE SHOVAR
    IF (RANGE(K1,1,26)) CALL EDERR(10)
    DO 211 I=1,5
211 IF (DSPVAR(I).EQ.K1) CALL EDITOR
    CONTINUE
    DO 215 I=1,5
    IF (DSPVAR(I).NE.0) GO TO 215
    CALL PUTVAR(I,K1)

```

```

CALL EDITOR
215 CONTINUE
CALL EDERR(11)
END

C
C
SUBROUTINE PUTVAR(I,J)
DSPVAR(I)=J
I=I+ORGVAL
TEMP(1)=EQUAL+LETTER(J) * 256
CALL BN2DEC(VAR(J),TEMP(2))
CALL MOVU2D(TEMP,BUTTON(I)+4,4)
CALL MOVU2D(INTON,BUTTON(I),1)
RETURN
END

C
C
SUBROUTINE REMVAR
DO 222 I=1,5
IF(DSPVAR(I).NE.K1) GO TO 222
N=I+ORGVAL
CALL MOVU2D(INTOFF,BUTTON(N),1)
DSPVAR(I)=0
CALL EDITOR
222 CONTINUE
CALL EDITOR
END

C
C
SUBROUTINE SAVE
DATA ISPACE,IDOL1/32,2H$1/
J=1
10 J=POINT(J)
IF(J.EQ.0) GO TO 20
IF(VIZ(J).EQ.3) GO TO 10
DO 5 I=1,80
5 INPUT(I)=ISPACE
CALL OUTLIN(J)
CALL WRLS(INPUT(1))
GO TO 10
20 INPUT(41)=IDOL1
CALL WRLS(INPUT(41))
CALL SEARCH(4,0,2)
CALL RESTRT
END

C
C
SUBROUTINE OUTLIN(LINE)
DO 10 I=2,11
INPUT(2*I+2)=STUFF(I,LINE)/256
10 INPUT(2*I+3)=STUFF(I,LINE)-INPUT(2*I+2)*256
INPUT(1)=STUFF(1,LINE)/256
INPUT(2)=STUFF(1,LINE)-INPUT(1)*256
DO 15 I=1,40
15 INPUT(I)=INPUT(2*I-1)*256+INPUT(2*I)+128-32767-1
RETURN
END

C
C
SUBROUTINE RESTRT
CALL REGEN(1)
CALL EDITOR
END

C
C
SUBROUTINE PAGE
I=LINNUM(SCRNSZ)
IF(POINT(I).EQ.0) CALL EDITOR
CALL REGEN(POINT(I))
CALL EDITOR
END

C
C

```

```

SUBROUTINE ROLL
  IF(K1.LE.0) CALL EDITOR
  IDUM=TOPLIN
  JDUM=SPEED
  SPEED = ENDSPD-2
  DO 20 I=1,K1
10    IDUM=POINT(IDUM)
      IF(IDUM.EQ.0) GO TO 30
      IF(VIZ(IDUM).EQ.3.OR.VIZ(IDUM).GT.4) GO TO 10
      CALL DELAY
      TOPLIN=IDUM
20    CALL REGEN(IDUM)
30    SPEED=JDUM
      CALL EDITOR
      END
C
C
      LOGICAL FUNCTION RANGE(I,J,K)
      RANGE=.TRUE.
      IF(I .GE. J .AND. I .LE. K) RANGE=.FALSE.
      RETURN
      END
C
C
      SUBROUTINE EDERR(N)
      WRITE(1,10)N
10    FORMAT(11H EDIT ERROR,I4)
      CALL EDITOR
      END
C
C
      SUBROUTINE RENTRY
      CALL INIT
      CALL EDITOR
      END

```

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
WASHINGTON, D. C. 20546
OFFICIAL BUSINESS

FIRST CLASS MAIL



POSTAGE AND FEES PAID
NATIONAL AERONAUTICS AND
SPACE ADMINISTRATION

POSTMASTER: If Undeliverable (Section 158
Postal Manual) Do Not Return

"The aeronautical and space activities of the United States shall be conducted so as to contribute . . . to the expansion of human knowledge of phenomena in the atmosphere and space. The Administration shall provide for the widest practicable and appropriate dissemination of information concerning its activities and the results thereof."

— NATIONAL AERONAUTICS AND SPACE ACT OF 1958

NASA SCIENTIFIC AND TECHNICAL PUBLICATIONS

TECHNICAL REPORTS: Scientific and technical information considered important, complete, and a lasting contribution to existing knowledge.

TECHNICAL NOTES: Information less broad in scope but nevertheless of importance as a contribution to existing knowledge.

TECHNICAL MEMORANDUMS: Information receiving limited distribution because of preliminary data, security classification, or other reasons.

CONTRACTOR REPORTS: Scientific and technical information generated under a NASA contract or grant and considered an important contribution to existing knowledge.

TECHNICAL TRANSLATIONS: Information published in a foreign language considered to merit NASA distribution in English.

SPECIAL PUBLICATIONS: Information derived from or of value to NASA activities. Publications include conference proceedings, monographs, data compilations, handbooks, sourcebooks, and special bibliographies.

TECHNOLOGY UTILIZATION PUBLICATIONS: Information on technology used by NASA that may be of particular interest in commercial and other non-aerospace applications. Publications include Tech Briefs, Technology Utilization Reports and Technology Surveys.

Details on the availability of these publications may be obtained from:

SCIENTIFIC AND TECHNICAL INFORMATION DIVISION
NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
Washington, D.C. 20546